
LOGIC AND COMPUTER DESIGN FUNDAMENTALS

THIRD EDITION

M. Morris Mano
California State University, Los Angeles

Charles R. Kime
University of Wisconsin, Madison

PEARSON
Prentice
Hall

Upper Saddle River, NJ 07458

Library of Congress Cataloging-in-Publication Data on file.

Vice President and Editorial Director, ECS: *Marcia J. Horton*

Publisher: *Tom Robbins*

Vice President and Director of Production and Manufacturing, ESM: *David W. Riccardi*

Executive Managing Editor: *Vince O'Brien*

Managing Editor: *David A. George*

Production Editor: *Daniel Sandin*

Director of Creative Services: *Paul Belfanti*

Creative Director: *Carole Anson*

Art Editor: *Xiaohong Zhu*

Manufacturing Manager: *Trudy Piscioti*

Manufacturing Buyer: *Lisa McDowell*

Marketing Manager: *Holly Stark*

About the Cover: The cover background is a computer-manipulated image by Charles R. Kime showing cascades on No Name Creek in the Siskiyou National Forest, Oregon. The cascades are symbolic of the top-down digital design process represented by the overlays of a block diagram, VHDL and Verilog code, and a gate implementation.



© 2004, 2001, 2000, 1997 Pearson Education, Inc.

Pearson Prentice Hall

Pearson Education, Inc.

Upper Saddle River, NJ 07458

All rights reserved. No part of this book may be reproduced, in any form or by any means, without permission in writing from the publisher.

Pearson Prentice Hall® is a trademark of Pearson Education, Inc.

Altera and Flex 10K are trademarks of Altera Corporation. GAL and PAL are trademarks of Lattice Semiconductor Corporation. Mentor Graphics, Model Technology, and ModelSim are trademarks of Mentor Graphics Corporation. RAMBUS and RDRAM are registered trademarks of RAMBUS, Inc. Xilinx and Spartan are registered trademarks of Xilinx, Inc. Verilog is a registered trademark of Cadence Design Systems, Inc. PowerPoint is registered trademark of the Microsoft Corporation.

The author and publisher of this book have used their best efforts in preparing this book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The author and publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The author and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

Printed in the United States of America

10 9 8 7 6 5 4 3 2

ISBN 0-13-140539-X

Pearson Education Ltd., *London*

Pearson Education Australia Pty. Ltd., *Sydney*

Pearson Education Singapore, Pte. Ltd.

Pearson Education North Asia Ltd., *Hong Kong*

Pearson Education Canada, Inc., *Toronto*

Pearson Educación de México, S.A. de C.V.

Pearson Education—Japan, *Tokyo*

Pearson Education Malaysia, Pte. Ltd.

Pearson Education, Inc., *Upper Saddle River, New Jersey*

CONTENTS

Preface	xiii
---------	------

□ Chapter 1 3

DIGITAL COMPUTERS AND INFORMATION	3
1-1 Digital Computers	3
Information Representation	5
Computer Structure	6
More on the Generic Computer	6
1-2 Number Systems	8
Binary Numbers	9
Octal and Hexadecimal Numbers	11
Number Ranges	13
1-3 Arithmetic Operations	13
Conversion from Decimal to Other Bases	16
1-4 Decimal Codes	18
BCD Addition	19
Parity Bit	20
1-5 Gray Codes	21
1-6 Alphanumeric Codes	23
ASCII Character Code	24
1-7 Chapter Summary	25
References	26
Problems	26

□ Chapter 2 29

COMBINATIONAL LOGIC CIRCUITS	29
2-1 Binary Logic and Gates	29
Binary Logic	30
Logic Gates	32
2-2 Boolean Algebra	33
Basic Identities of Boolean Algebra	35
Algebraic Manipulation	37

	Complement of a Function	40
2-3	Standard Forms	41
	Minterms and Maxterms	42
	Sum of Products	45
	Product of Sums	46
2-4	Two-Level Circuit Optimization	47
	Cost Criteria	48
	Two-Variable Map	49
	Three-Variable Map	50
	Four-Variable Map	55
2-5	Map Manipulation	58
	Essential Prime Implicants	59
	Nonessential Prime Implicants	61
	Product-of-Sums Optimization	62
	Don't-Care Conditions	63
2-6	Multiple-Level Circuit Optimization	65
2-7	Other Gate Types	70
2-8	Exclusive-OR Operator and Gates	75
	Odd Function	76
2-9	High-Impedance Outputs	77
2-10	Chapter Summary	80
	References	80
	Problems	81

□ Chapter 3 **87**

	COMBINATIONAL LOGIC DESIGN	87
3-1	Design Concepts and Automation	87
	Design Hierarchy	89
	Top-Down Design	92
	Computer-Aided Design	92
	Hardware Description Languages	93
	Logic Synthesis	95
3-2	The Design Space	96
	Gate Properties	97
	Levels of Integration	97
	Circuit Technologies	97
	Technology Parameters	98
	Positive and Negative Logic	101
	Design Trade-Offs	103
3-3	Design Procedure	104
3-4	Technology Mapping	110
	Cell Specification	112

	Libraries	112
	Mapping Techniques	113
3-5	Verification	121
	Manual Logic Analysis	121
	Simulation	122
3-6	Programmable Implementation Technologies	124
	Read-Only Memory	127
	Programmable Logic Array	129
	Programmable Array Logic Devices	130
3-7	Chapter Summary	132
	References	133
	Problems	133

□ Chapter 4 **141**

	COMBINATIONAL FUNCTIONS AND CIRCUITS	141
4-1	Combinational Circuits	141
4-2	Rudimentary Logic Functions	142
	Value-Fixing, Transferring, and Inverting	142
	Multiple-Bit Functions	143
	Enabling	146
4-3	Decoding	147
	Decoder Expansion	148
	Decoder and Enabling Combinations	151
4-4	Encoding	152
	Priority Encoder	153
	Encoder Expansion	155
4-5	Selecting	156
	Multiplexers	156
	Multiplexer Expansion	158
	Alternative Selection Implementations	159
4-6	Combinational Function Implementation	161
	Using Decoders	162
	Using Multiplexers	164
	Using Read-Only Memories	166
	Using Programmable Logic Arrays	169
	Using Programmable Array Logic Devices	171
	Using Lookup Tables	175
4-7	HDL Representation for Combinational Circuits—VHDL	176
4-8	HDL Representations for Combinational Circuits—Verilog	184
4-9	Chapter Summary	190
	References	191
	Problems	191

ARITHMETIC FUNCTIONS AND CIRCUITS	201
5-1 Iterative Combinational Circuits	201
5-2 Binary Adders	202
Half Adder	203
Full Adder	204
Binary Ripple Carry Adder	205
Carry Lookahead Adder	206
5-3 Binary Subtraction	210
Complements	212
Subtraction with Complements	213
5-4 Binary Adder–Subtractors	215
Signed Binary Numbers	216
Signed Binary Addition and Subtraction	218
Overflow	220
5-5 Binary Multiplication	221
5-6 Other Arithmetic Functions	223
Contraction	224
Incrementing	225
Decrementing	226
Multiplication by Constants	227
Division by Constants	227
Zero Fill and Extension	227
5-7 HDL Representations—VHDL	229
Behavioral Description	231
5-8 HDL Representations—Verilog	233
Behavioral Description	234
5-9 Chapter Summary	235
References	235
Problems	236

SEQUENTIAL CIRCUITS	241
6-1 Sequential Circuit Definitions	242
6-2 Latches	244
SR and $\overline{S} \overline{R}$ Latches	245
D Latch	248
6-3 Flip-Flops	249
Master-Slave Flip-Flops	250
Edge-Triggered Flip-Flop	253
Standard Graphics Symbols	254

	Direct Inputs	256
	Flip-Flop Timing	257
6-4	Sequential Circuit Analysis	258
	Input Equations	258
	State Table	260
	State Diagram	262
	Sequential Circuit Timing	263
	Simulation	266
6-5	Sequential Circuit Design	267
	Design Procedure	268
	Finding State Diagrams and State Tables	268
	State Assignment	275
	Designing with <i>D</i> Flip-Flops	275
	Designing with Unused States	277
	Verification	279
6-6	Other Flip-Flop Types	281
	JK and T Flip-Flops	282
6-7	HDL Representation for Sequential Circuits—VHDL	284
6-8	HDL Representation for Sequential Circuits—Verilog	291
6-9	Chapter Summary	298
	References	299
	Problems	299

□ Chapter 7 **309**

	REGISTERS AND REGISTER TRANSFERS	309
7-1	Registers and Load Enable	310
	Register with Parallel Load	311
7-2	Register Transfers	313
7-3	Register Transfer Operations	315
7-4	A Note for VHDL and Verilog Users Only	318
7-5	Microoperations	318
	Arithmetic Microoperations	319
	Logic Microoperations	321
	Shift Microoperations	323
7-6	Microoperations on a Single Register	324
	Multiplexer-Based Transfers	324
	Shift Registers	326
	Ripple Counter	331
	Synchronous Binary Counters	333
	Other Counters	337
7-7	Register Cell Design	339
7-8	Multiplexer and Bus-Based Transfers for Multiple Registers	345
	Three-State Bus	346

7-9	Serial Transfer and Microoperations	348
	Serial Addition	349
7-10	HDL Representation for Shift Registers and Counters —VHDL	351
7-11	HDL Representation for Shift Registers and Counters—Verilog	353
7-12	Chapter Summary	354
	References	356
	Problems	356

□ Chapter 8 **363**

SEQUENCING AND CONTROL		363
8-1	The Control Unit	364
8-2	Algorithmic State Machines	365
	The ASM Chart	365
	Timing Considerations	368
8-3	ASM Chart Examples	369
	Binary Multiplier	369
8-4	Hardwired Control	375
	Sequence Register and Decoder	378
	One Flip-Flop per State	380
8-5	HDL Representation of the Binary Multiplier—VHDL	384
8-6	HDL Representation of the Binary Multiplier—Verilog	387
8-7	Microprogrammed Control	390
8-8	Chapter Summary	392
	References	392
	Problems	393

□ Chapter 9 **399**

MEMORY BASICS		399
9-1	Memory Definitions	399
9-2	Random-Access Memory	400
	Write and Read Operations	402
	Timing Waveforms	402
	Properties of Memory	405
9-3	SRAM Integrated Circuits	405
	Coincident Selection	408
9-4	Array of SRAM ICs	411
9-5	DRAM ICs	415
	DRAM Cell	415
	DRAM Bit Slice	417
9-6	DRAM Types	421
	Synchronous DRAM (SDRAM)	421

	Double Data Rate SDRAM (DDR SDRAM)	424
	RAMBUS® DRAM (RDRAM)	425
9-7	Arrays of Dynamic RAM ICs	426
9-8	Chapter Summary	426
	References	427
	Problems	427

□ Chapter 10 **429**

COMPUTER DESIGN BASICS		429
10-1	Introduction	430
10-2	Datapaths	430
10-3	The Arithmetic/Logic Unit	433
	Arithmetic Circuit	434
	Logic Circuit	437
	Arithmetic/Logic Unit	437
10-4	The Shifter	439
	Barrel Shifter	440
10-5	Datapath Representation	441
10-6	The Control Word	444
10-7	A Simple Computer Architecture	449
	Instruction Set Architecture	450
	Storage Resources	450
	Instruction Formats	451
	Instruction Specifications	454
10-8	Single-Cycle Hardwired Control	456
	Instruction Decoder	458
	Sample Instructions and Program	460
	Single-Cycle Computer Issues	462
10-9	Multiple-Cycle Hardwired Control	464
	Sequential Control Design	467
10-10	Chapter Summary	475
	References	475
	Problems	476

□ Chapter 11 **483**

INSTRUCTION SET ARCHITECTURE		483
11-1	Computer Architecture Concepts	483
	Basic Computer Operation Cycle	484
	Register Set	485
11-2	Operand Addressing	485
	Three-Address Instructions	486

	Two-Address Instructions	487
	One-Address Instructions	487
	Zero-Address Instructions	488
	Addressing Architectures	489
11-3	Addressing Modes	492
	Implied Mode	493
	Immediate Mode	493
	Register and Register-Indirect Modes	493
	Direct Addressing Mode	494
	Indirect Addressing Mode	496
	Relative Addressing Mode	496
	Indexed Addressing Mode	496
	Summary of Addressing Modes	497
11-4	Instruction Set Architectures	499
11-5	Data Transfer Instructions	500
	Stack Instructions	501
	Independent versus Memory-Mapped I/O	502
11-6	Data Manipulation Instructions	503
	Arithmetic Instructions	504
	Logical and Bit Manipulation Instructions	505
	Shift Instructions	506
11-7	Floating-Point Computations	507
	Arithmetic Operations	508
	Biased Exponent	509
	Standard Operand Format	510
11-8	Program Control Instructions	512
	Conditional Branch Instructions	513
	Procedure Call and Return Instructions	516
11-9	Program Interrupt	517
	Types of Interrupts	518
	Processing External Interrupts	519
11-10	Chapter Summary	521
	References	521
	Problems	522

□ Chapter 12 **527**

	RISC AND CISC CENTRAL PROCESSING UNITS	527
12-1	Pipelined Datapath	528
	Execution of Pipeline Microoperations	532
12-2	Pipelined Control	533
	Pipeline Programming and Performance	535
12-3	The Reduced Instruction Set Computer	537

	Instruction Set Architecture	538
	Addressing Modes	541
	Datapath Organization	541
	Control Organization	544
	Data Hazards	546
	Control Hazards	553
12-4	The Complex Instruction Set Computer	557
	ISA Modifications	559
	Datapath Modifications	560
	Control Unit Modifications	562
	Microprogrammed Control	563
	Microprograms for Complex Instructions	566
12-5	More on Design	569
	High-Performance CPU Concepts	569
	Recent Architectural Innovations	572
	Digital Systems	573
12-6	Chapter Summary	574
	References	575
	Problems	575

□ Chapter 13 **579**

	INPUT-OUTPUT AND COMMUNICATION	579
13-1	Computer I/O	579
13-2	Sample Peripherals	580
	Keyboard	580
	Hard Disk	581
	Graphics Display	583
	I/O Transfer Rates	584
13-3	I/O Interfaces	584
	I/O Bus and Interface Unit	585
	Example of I/O Interface	586
	Strobing	588
	Handshaking	589
13-4	Serial Communication	591
	Asynchronous Transmission	592
	Synchronous Transmission	593
	The Keyboard Revisited	593
	A Packet-Based Serial I/O Bus	594
13-5	Modes of Transfer	598
	Example of Program-Controlled Transfer	599
	Interrupt-Initiated Transfer	601
13-6	Priority Interrupt	601

	Daisy Chain Priority	602
	Parallel Priority Hardware	603
13-7	Direct Memory Access	605
	DMA Controller	606
	DMA Transfer	608
13-8	I/O Processors	609
13-9	Chapter Summary	612
	References	613
	Problems	613

□ Chapter 14 **617**

	MEMORY SYSTEMS	617
14-1	Memory Hierarchy	618
14-2	Locality of Reference	620
14-3	Cache Memory	622
	Cache Mappings	624
	Line Size	629
	Cache Loading	631
	Write Methods	631
	Integration of Concepts	632
	Instruction and Data Caches	635
	Multiple-Level Caches	635
14-4	Virtual Memory	636
	Page Tables	638
	Translation Lookaside Buffer	640
	Virtual Memory and Cache	642
14-5	Chapter Summary	642
	References	643
	Problems	643
	INDEX	647

PREFACE

The object of this text is to provide an understanding of the fundamentals of logic and computer design for a wide audience of readers. Many of the fundamentals that are presented here have not changed in decades. On the other hand, the advances in underlying technology have had a major effect on the application of these fundamentals and the emphasis to be placed on them. The process of design has been automated by using hardware description languages and logic synthesis, and the quest for speed and low power has changed the fundamentals of computer design.

The content of this third edition continues to focus on fundamentals while at the same time reflecting the relative importance of basic concepts as the technology and the design process evolve. As an illustration, microprogramming, which has declined in use as a primary control unit design method, is treated only as a control unit design technique for implementing complex computer instructions. Also, over time, the fundamental terminology is evolving and, along with it, our perspective of associated concepts. For example, in this edition, sections on NAND circuits and NOR circuits appear in the broader context of technology mapping.

The text continues to provide the option to instructors to provide very basic coverage of either VHDL or Verilog® or omit hardware description language (HDL) coverage entirely. The perspective of the introductory coverage here is the correspondence of the HDL description to the actual hardware it represents. This vital perspective, which is critical in writing HDLs for logic synthesis, can be lost in more detailed treatments focusing on the language and fluency in its use.

In summary, this edition of *Logic and Computer Design Fundamentals* features a strong emphasis on fundamentals underlying contemporary logic design using hardware description languages, synthesis, and verification as well as changes in emphasis in the use of fundamentals of computer design. The focus on basic concepts remains and manual exercises to enhance thorough understanding of these concepts continue as a mainstay.

In order to support the evolving perspective and to deal with growing structural problems, notably chapter length, this edition features a major chapter reorganization. Chapters 1 through 6 of the book treat logic design, and Chapters 7 through 9 deal with digital systems design. Chapters 10 through 14 focus directly on computer design. This arrangement provides solid digital system design fundamentals while accomplishing a gradual, bottom-up development of fundamentals for use in top-down computer design in later chapters. Eleven of the 14 chapters

contain new material not present in the second edition, and approximately 50% of the problems are modified or new. There are over a dozen text supplements available on the text website, which represent both new material and deleted material from prior editions. Summaries of the topics covered in each chapter follow.

Chapter 1—Digital Computers and Information introduces computer systems and information representation, including a new section on Gray codes.

Chapter 2—Combinational Logic Circuits deals with the basic theory and concepts for designing and optimizing gate circuits. A new section on multi-level logic optimization appears. In addition to the basic literal count, gate input count is introduced as a more accurate cost criterion for use with multilevel logic circuits.

Chapter 3—Combinational Logic Design provides an overview of the contemporary logic design process and deals with gate characteristics and delay, and technology issues such as use of NAND, NOR, AOI and OAI, and XOR and XNOR gate functions. The details of steps of the design process including problem formulation, logic optimization, technology mapping, and verification are covered for combinational logic. As a part of technology mapping, this chapter contains basic coverage of ROMs, PLAs, and PALs. Coverage of Field Programmable Gate Arrays (FPGAs), with the focus on the parts typically used in student labs, is provided as a website supplement to permit updating as this technology changes during the lifetime of this edition.

Chapter 4—Combinational Functions and Circuits covers the building blocks of combinational design. Remnants of MSI logic have been removed with the focus changed to 1) fundamental combinational functions and their implementations and 2) techniques for utilizing and modifying these functions and their associated implementations. This focus provides fundamentals for a clearer understanding of structured logic design and for visualizing the logic resulting from HDL synthesis. In addition to covering decoding, encoding, code conversion, selecting, and distributing, new functions such as enabling and input-fixing are introduced. Introductory sections on Verilog and VHDL are provided for the various types of functions.

Chapter 5—Arithmetic Functions and Circuits deals with arithmetic functions and their implementations. Beyond number representation, addition, subtraction and multiplication, functions and implementations are introduced for incrementing, decrementing, filling, extension and shifting. Verilog and VHDL descriptions are provided for arithmetic functions.

Chapter 6—Sequential Circuits introduces sequential circuit analysis and design. Latches, master-slave flip-flops, and edge-triggered flip-flops are covered with an emphasis on the D type. Other types of flip-flops (S-R, J-K, and T), which are used less frequently in modern designs, are covered but given less emphasis with in-depth coverage moved to a website supplement. Verilog and VHDL descriptions of flip-flops and sequential circuits are provided.

Chapter 7—Registers and Register Transfers ties together closely the implementation of registers and their applications. Shift register and counter design are based on combining registers with functions and implementations introduced in Chapters 4 and 5. Only the ripple counter is presented as a totally new concept. This approach is in keeping with the reduction of focus on circuits originating as

MSI parts. A new section focuses on register cell design for registers performing multiple operations. Verilog and VHDL descriptions of the various register types are introduced.

Chapter 8—Sequencing and Control covers control unit design. An additional feature added to the Algorithmic State Machine (ASM) representation is a multiway branch that is analogous to the “case” from Verilog and VHDL. Hardware control is emphasized, with reduced emphasis on microprogrammed control.

Chapter 9—Memory Basics covers SRAM, DRAM, and basic memory systems. A new section on synchronous DRAMs treats the basics of these current technologies. Verilog and VHDL memory models are provided on the website.

Chapter 10—Computer Design Basics covers register files, function units, datapaths, and two simple computers. A single-cycle computer and a new multiple-cycle computer are designed in some detail with both employing hard-wired control.

Chapter 11—Instruction Set Architecture introduces many facets of instruction set architecture. It deals with address count, addressing modes, architectures, and the types of instructions. Addressing modes and other aspects of instructions are illustrated with brief segments of instruction code.

Chapter 12—RISC and CISC Central Processing Units introduces pipelined datapaths and control. A Reduced Instruction Set Computer (RISC) design using pipelining is given. A new Complex Instruction Set Computer (CISC) design is presented. This design uses a microprogrammed control unit in conjunction with the RISC foundation to implement complex instructions.

Chapter 13—Input-Output and Communication deals with data transfer between the CPU, input-output interfaces and peripheral devices. Discussions of a keyboard, a CRT display, and a hard disk as peripherals are included, and a keyboard interface is illustrated. Other topics covered range from serial communication, including the Universal Serial Bus (USB) as an illustration, to I/O processors.

Chapter 14—Memory Systems has a particular focus on memory hierarchies. The concept of locality of reference is introduced and illustrated by consideration of the cache/main memory and main memory/hard disk relationships. An overview of cache design parameters is provided. The treatment of memory management focuses on paging and a translation lookaside buffer supporting virtual memory.

In addition to the text itself, there are substantial support features provided, descriptions of which follow.

Companion Website (<http://www.prenhall.com/mano>) content includes the following material: 1) twelve reading supplements including new material and material deleted from prior editions, 2) VHDL and Verilog source files for all examples, 3) solutions for about one-third of all text Chapter and reading supplement problems, 4) errata, 5) PowerPoint® slides for Chapters 1 through 9, and 6) projection originals for complex figures and tables from the text.

Design Tools packaged with most domestic and international printings of the text consist of the Xilinx® ISE Student Edition software graciously provided without charge by Xilinx, Inc. Download access to the demo XE version of the ModelSim® logic simulator from Model Technology Incorporated is available

through Xilinx. These tools can be used to enter schematics and state diagrams, compile and simulate VHDL code, Verilog code, or schematics, synthesize CPLD and FPGA implementations and simulate the resulting implementations. With the purchase of low-cost experimental hardware, these tools provide everything needed for students to perform CPLD-based or FPGA-based experiments.

Instructor's Manual content includes important suggestions for use of the book, information for obtaining alternative CAD tools, and all problem solutions. This manual is available from Prentice Hall to instructors at academic institutions who adopt the book for classroom use.

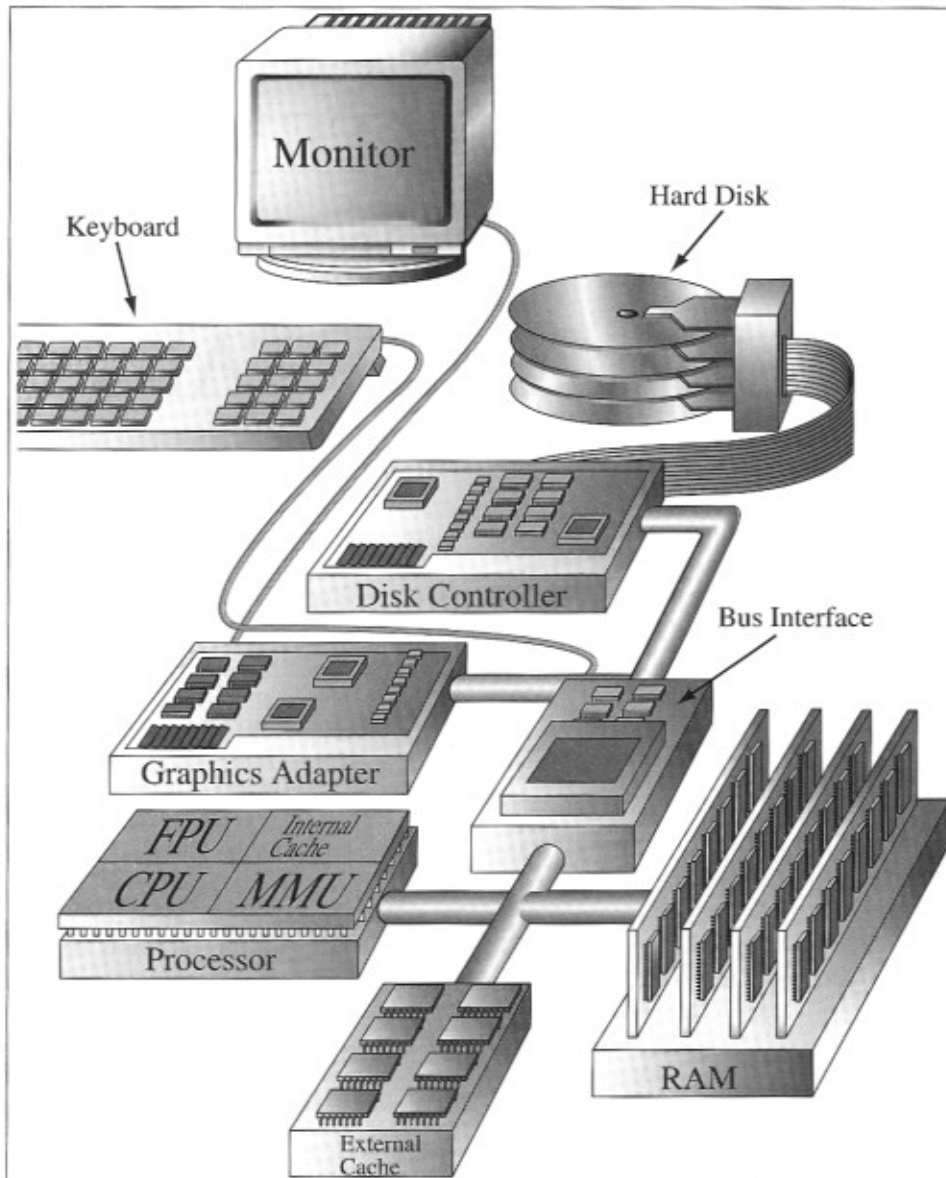
Because of its broad coverage of both logic and computer design, this book can serve several different objectives in sophomore through junior level courses. Chapters 1 through 11, with selected sections omitted, provide an overview of hardware for computer science, computer engineering, electrical engineering or engineering students in general in a single semester course. Chapters 1 through 8 give a basic introduction to logic design, which can be completed in a single quarter for electrical and computer engineering students. Coverage of Chapters 1 through 10 in a semester, with perhaps some supplementary material, provides a stronger, more contemporary logic design treatment. The entire book, covered in two quarters, provides the basics of logic and computer design for computer engineering and science students. Coverage of the entire book with appropriate supplementary material or a laboratory component can fill a two-semester sequence in logic design and computer architecture. Finally, due to its moderately paced treatment of a wide range of topics, the book is ideal for self-study by engineers and computer scientists.

Among the many contributions to this book, excellent detailed comments and suggestions on drafts of Chapters 1 through 8 were provided by Richard E. Haskell, Oakland University; Eugene Henry, University of Notre Dame; Sung Hu, San Francisco State University; and Walid Hubbi, New Jersey Institute of Technology. Their contributions to text improvements are greatly appreciated. Faculty and students at the University of Wisconsin also contributed to the text. A suggestion from Professor Jim Smith motivated the direction taken in CISC design in Chapter 12, and Professor Leon Shohet suggested specific improvements based on his use of the 2nd Edition. A special thanks goes to Eric Weglarz for his in-depth reviews of new material for both content and clarity. Also thanks to Eric and to Jim Liu for preparing solutions to new and modified problems for the Instructor's Manual. Our special appreciation goes out to all of those at Prentice Hall and elsewhere for their efforts on this edition. Notable are Tom Robbins and Alice Dworkin for their guidance and support, Eric Frank for his contributions during the early stages of this edition, and Daniel Sandin for his very efficient and helpful handling of the production of this edition.

Finally, a very special thanks to Val Kime for her patience and understanding throughout the development of the third edition.

M. MORRIS MANO
CHARLES R. KIME

LOGIC AND
COMPUTER
DESIGN
FUNDAMENTALS



CHAPTER

1

DIGITAL COMPUTERS AND INFORMATION

Logic design fundamentals and computer design fundamentals are the topics of this book. Logic design deals with the basic concepts and tools used to design digital hardware consisting of logic circuits. Computer design deals with the additional concepts and tools used to design computers and other complex digital hardware. Computers and digital hardware in general are referred to as digital systems. Thus, this book is about understanding and designing digital systems. Due to its generality and complexity, the computer provides an ideal vehicle for learning the concepts of, and tools for, digital system design. In addition, due to its widespread use, the computer itself is deserving of study. Hence, the focus in this book is on computers and their design.

The computer will be not only a vehicle, but also a motivator for study. To this end, we use the exploded pictorial diagram of a computer of the class commonly referred to as a PC (personal computer) given on the opposite page. We use this generic computer to highlight the significance of the material covered and its relationship to the overall system. A bit later in the chapter, we will discuss the various major components of the generic computer and see how they relate to a block diagram often used to describe a computer.

1-1 DIGITAL COMPUTERS

Today, digital computers have such a prominent and growing role in modern society that we often say we are in the “information age.” Computers are involved in our business transactions, communications, transportation, medical treatment, and entertainment. They monitor our weather and environment. In the industrial world,

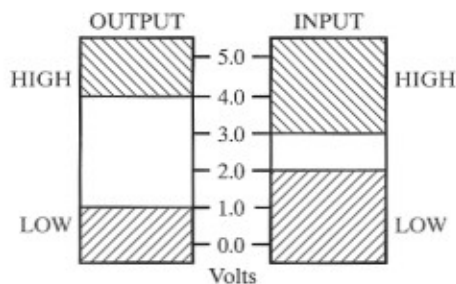
they are heavily employed in design, manufacturing, distribution, and sales. They have contributed to many scientific discoveries and engineering developments that would have been unattainable otherwise. Notably, the design of a new processor for a modern computer could not be done without the use of many computers!

The most striking property of the digital computer is its generality. It can follow a sequence of instructions, called a program, that operates on given data. The user can specify and change the program or the data according to specific needs. As a result of this flexibility, general-purpose digital computers can perform a variety of information-processing tasks that range over a very wide spectrum of applications. The general-purpose digital computer is the best known example of a *digital system*. Characteristic of a digital system is its manipulation of discrete elements of information. Any set that is restricted to a finite number of elements contains discrete information. Examples of discrete sets are the 10 decimal digits, the 26 letters of the alphabet, the 52 playing cards, and the 64 squares of a chessboard. Early digital computers were used mostly for numeric computations. In this case, the discrete elements used were the digits. From such an application, the term *digital computer* emerged.

Discrete elements of information are represented in a digital system by physical quantities called *signals*. Electrical signals such as voltages and currents are most common. Electronic devices called transistors predominate in the circuitry that implements these signals. The signals in most present-day electronic digital systems use just two discrete values and are therefore said to be *binary*.

We typically represent the two discrete values by ranges of voltage values called HIGH and LOW. Output voltage ranges and input voltage ranges are illustrated in Figure 1-1. The HIGH output voltage value ranges between 4.0 and 5.5 volts, and the LOW output voltage value ranges between -0.5 and 1.0 volt. The high input range allows 3.0 to 5.5 volts to be recognized as a HIGH, and the low input range allows -0.5 to 2.0 volts to be recognized as a LOW. The fact that the input ranges are longer than the output ranges allows the circuits to function correctly in spite of variations in their behavior and undesirable “noise” voltages that may be added to or subtracted from the outputs.

We give the output and input voltage ranges a number of different names. Among these are HIGH(H) and LOW(L), TRUE(T) and FALSE(F), and 1 and 0.



□ **FIGURE 1-1**
An Example of Voltage Ranges for Binary Signals

It is clear that the higher voltage ranges are associated with HIGH or H, and the lower voltage ranges with LOW or L. We find, however, that for TRUE and 1 and FALSE and 0, there is a choice. TRUE and 1 can be associated with either the higher or lower voltage range and FALSE and 0 with the other range. Unless otherwise indicated, we assume that TRUE and 1 are associated with the higher of the voltage ranges, H, and that FALSE and 0 are associated with the lower of the voltage ranges, L.

Why is binary used? In contrast to the situation in Figure 1-1, consider a system with 10 values representing the decimal digits. In such a system, the voltages available—say, 0 to 5.0 volts—could be divided into 10 ranges, each of length 0.5 volt. A circuit would provide an output voltage within each of these 10 ranges. An input of a circuit would need to determine in which of the 10 ranges an applied voltage lies. If we wish to allow for noise on the voltages, then output voltage might be permitted to range over less than 0.25 volt for a given digit representation, and boundaries between inputs could vary by only less than 0.25 volt. This would require complex and costly electronic circuits and still could be disturbed by small “noise” voltages or small variations in the circuits occurring during their manufacture or use. As a consequence, the use of such multivalued circuits is very limited. Instead, binary circuits are used in which correct circuit operation can be achieved with significant variations in both the two output voltages and the two input ranges. The resulting transistor circuit with an output that is either HIGH or LOW is simple, easy to design, and extremely reliable.

Information Representation

Since 0 and 1 are associated with the binary number system, they are the preferred names for the signal ranges. A binary digit is called a *bit*. Information is represented in digital computers by groups of bits. By using various coding techniques, groups of bits can be made to represent not only binary numbers, but also other groups of discrete symbols. Groups of bits, properly arranged, can even specify to the computer the instructions to be executed and the data to be processed.

Discrete quantities of information either emerge from the nature of the data being processed or may be purposely quantized from continuous values. For example, a payroll schedule is inherently discrete data containing employee names, social security numbers, weekly salaries, income taxes, and so on. An employee's paycheck is processed using discrete data values such as letters of the alphabet (for the employee's name), digits (for the salary), and special symbols such as \$. On the other hand, an engineer may measure the speed of rotation of an automobile wheel, which varies continuously with time, but may record only specific values at specific times in tabular form. The engineer is thus quantizing the continuous data, making each number in the table a discrete quantity of information. In a case such as this, if the measurement can be converted to an electronic signal, the quantization of the signal in both value and time can be performed automatically by an analog-to-digital conversion device.

Computer Structure

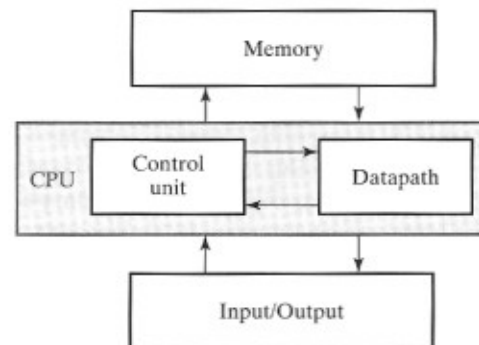
A block diagram of a digital computer is shown in Figure 1-2. The memory stores programs as well as input, output, and intermediate data. The datapath performs arithmetic and other data-processing operations as specified by the program. The control unit supervises the flow of information between the various units. A datapath, when combined with the control unit, forms a component referred to as a *central processing unit*, or CPU.

The program and data prepared by the user are transferred into memory by means of an input device such as a keyboard. An output device, such as a CRT (cathode-ray tube) monitor, displays the results of the computations and presents them to the user. A digital computer can accommodate many different input and output devices, such as hard disks, floppy disk drives, CD-ROM drives, and scanners. These devices use some digital logic, but often include analog electronic circuits, optical sensors, CRTs or LCDs (liquid crystal displays), and electro-mechanical components.

The control unit in the CPU retrieves the instructions, one by one, from the program stored in the memory. For each instruction, the control unit manipulates the datapath to execute the operation specified by the instruction. Both program and data are stored in memory. A digital computer is a powerful system. It can perform arithmetic computations, manipulate strings of alphabetic characters, and be programmed to make decisions based on internal and external conditions.

More on the Generic Computer

At this point, we will briefly discuss the generic computer and relate its various parts to the block diagram in Figure 1-2. At the lower left of the diagram at the beginning of this chapter is the heart of the computer, an integrated circuit called the *processor*. Modern processors such as this one are quite complex and consist of millions of transistors. The processor contains four functional modules: the CPU, the FPU, the MMU, and the internal cache.



□ **FIGURE 1-2**
Block Diagram of a Digital Computer

We have already discussed the CPU. The FPU (floating-point unit) is somewhat like the CPU, except that its datapath and control unit are specifically designed to perform floating-point operations. In essence, these operations process information represented in the form of scientific notation (e.g., 1.234×10^7), permitting the generic computer to handle very large and very small numbers. The CPU and the FPU, in relation to Figure 1-2, each contain a datapath and a control unit.

The MMU is the memory management unit. The MMU plus the internal cache and the separate blocks near the bottom of the computer labeled “External Cache” and “RAM” (random access memory) are all part of the memory in Figure 1-2. The two caches are special kinds of memory that allow the CPU and FPU to get at the data to be processed much faster than with RAM alone. RAM is what is most commonly referred to as memory. As its main function, the MMU causes the memory that appears to be available to the CPU to be much, much larger than the actual size of the RAM. This is accomplished by data transfers between the RAM and the hard disk shown at the top of the picture of the generic computer. So the hard disk, which we discuss later as an input/output device, appears conceptually as a part of the memory and input/output.

The connection paths shown between the processor, memory, and external cache are the pathways between integrated circuits. These are typically implemented as fine copper conductors on a printed circuit board. The connection paths below the bus interface are referred to as the processor bus. The connections above the bus interface are referred to as the input/output (I/O) bus. The processor bus and the I/O bus attached to the bus interface carry data having different numbers of bits and have different ways of controlling the movement of data. They may also operate at different speeds. The bus interface hardware handles these differences so that data can be communicated between the two buses.

All of the remaining structures in the generic computer are considered part of I/O in Figure 1-2. In terms of sheer physical volume, these structures dominate. In order to enter information into the computer, a keyboard is provided. In order to view output in the form of text or graphics, a graphics adapter card and CRT monitor are provided. The hard disk, discussed previously, is an electromechanical magnetic storage device. It stores large quantities of information in the form of magnetic flux on spinning disks coated with magnetic materials. In order to control the hard disk and transfer information to and from it, a disk controller is used. The keyboard, graphics adapter card, and disk controller card are all attached to the I/O bus. This allows these devices to communicate through the bus interface with the CPU and other circuitry connected to the processor buses.

The generic computer consists mainly of an interconnection of digital modules. To understand the operation of each module, it is necessary to have a basic knowledge of digital systems and their general behavior. Chapters 1 through 6 of this book deal with logic design of digital circuits in general. Chapters 7 and 8 discuss the primary components of a digital system, their operation, and their design. The operational characteristics of RAM are explained in Chapter 9. Datapath and control for simple computers are introduced in Chapter 10. Chapters 11 through 14 present the basics of computer design. Typical instructions employed in computer instruction set

architectures are presented in Chapter 11. The architecture and design of CPUs are examined in Chapter 12. Input and output devices and the various ways that a CPU can communicate with them are discussed in Chapter 13. Finally, memory hierarchy concepts related to the caches and MMU are introduced in Chapter 14.

To guide the reader through this material and to keep in mind the “forest” as we carefully examine many of the “trees,” accompanying discussion appears in a blue box at the beginning of each chapter to tie the topics in the chapter to the associated components in the generic computer diagram at the start of this chapter. At the completion of our journey, we will have covered most of the various modules of the computer and will have an understanding of the fundamentals that underlie both its function and design.

Earlier, we mentioned that a digital computer manipulates discrete elements of information and that all information in the computer is represented in binary form. Operands used for calculations may be expressed in the binary number system or in the decimal system by means of a binary code. The letters of the alphabet are also converted into a binary code. The purpose of the remainder of this chapter is to introduce the binary number system, binary arithmetic, and selected binary codes as a basis for further study in the succeeding chapters. In relation to the generic computer, this material is very important and spans all of the components except some in I/O that involve mechanical operations and analog (as contrasted with digital) electronics.

1-2 NUMBER SYSTEMS

The decimal number system is employed in everyday arithmetic to represent numbers by strings of digits. Depending on its position in the string, each digit has an associated value of an integer raised to the power of 10. For example, the decimal number 724.5 is interpreted to represent 7 hundreds plus 2 tens plus 4 units plus 5 tenths. The hundreds, tens, units, and tenths are powers of 10 implied by the position of the digits. The value of the number is computed as follows:

$$724.5 = 7 \times 10^2 + 2 \times 10^1 + 4 \times 10^0 + 5 \times 10^{-1}$$

The convention is to write only the digits and infer the corresponding powers of 10 from their positions. In general, a decimal number with n digits to the left of the decimal point and m digits to the right of the decimal point is represented by a string of coefficients:

$$A_{n-1}A_{n-2}\dots A_1A_0.A_{-1}A_{-2}\dots A_{-m+1}A_{-m}$$

Each coefficient A_i is one of 10 digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9). The subscript value i gives the position of the coefficient and, hence, the weight 10^i by which the coefficient must be multiplied.

The decimal number system is said to be of *base* or *radix* 10, because the coefficients are multiplied by powers of 10 and the system uses 10 distinct digits. In

general, a number in base r contains r digits, 0, 1, 2, ..., $r - 1$, and is expressed as a power series in r with the general form

$$A_{n-1}r^{n-1} + A_{n-2}r^{n-2} + \dots + A_1r^1 + A_0r^0 \\ + A_{-1}r^{-1} + A_{-2}r^{-2} + \dots + A_{-m+1}r^{-m+1} + A_{-m}r^{-m}$$

When the number is expressed in positional notation, only the coefficients and the radix point are written down:

$$A_{n-1}A_{n-2}\dots A_1A_0.A_{-1}A_{-2}\dots A_{-m+1}A_{-m}$$

In general, the “.” is called the *radix point*. A_{n-1} is referred to as the *most significant digit* (msd), and A_{-m} is referred to as the *least significant digit* (lsd) of the number. Note that if $m = 0$, the lsd is $A_{-0} = A_0$. To distinguish between numbers of different bases, it is customary to enclose the coefficients in parentheses and place a subscript after the right parenthesis to indicate the base of the number. However, when the context makes the base obvious, it is not necessary to use parentheses. The following illustrates a base-5 number with $n = 3$ and $m = 1$ and its conversion to decimal:

$$(312.4)_5 = 3 \times 5^2 + 1 \times 5^1 + 2 \times 5^0 + 4 \times 5^{-1} \\ = 75 + 5 + 2 + 0.8 = (82.8)_{10}$$

Note that for all the numbers without the base designated, the arithmetic is performed with decimal numbers. Note also that the base-5 system uses only five digits, and, therefore, the values of the coefficients in a number can be only 0, 1, 2, 3, and 4 when expressed in that system.

An alternative method for conversion to base 10 that reduces the number of operations is based on a factored form of the power series:

$$(\dots((A_{n-1}r + A_{n-2})r + A_{n-3})r + \dots + A_1)r + A_0 \\ + (A_{-1} + (A_{-2} + (A_{-3} + \dots + (A_{-m+2} + (A_{-m+1} + A_{-m}r^{-1})r^{-1})r^{-1}\dots)r^{-1})r^{-1})r^{-1}$$

For the example above,

$$(312.4)_5 = ((3 \times 5 + 1) \times 5) + 2 + 4 \times 5^{-1} \\ = 16 \times 5 + 2 + 0.8 = (82.8)_{10}$$

In addition to decimal, three number systems are used in computer work: binary, octal, and hexadecimal. These are base-2, base-8, and base-16 number systems, respectively.

Binary Numbers

The binary number system is a base-2 system with two digits: 0 and 1. A binary number such as 11010.11 is expressed with a string of 1's and 0's and, possibly, a

□ **TABLE 1-1**
Powers of Two

n	2 ⁿ	n	2 ⁿ	n	2 ⁿ
0	1	8	256	16	65,536
1	2	9	512	17	131,072
2	4	10	1,024	18	262,144
3	8	11	2,048	19	524,288
4	16	12	4,096	20	1,048,576
5	32	13	8,192	21	2,097,152
6	64	14	16,384	22	4,194,304
7	128	15	32,768	23	8,388,608

binary point. The decimal equivalent of a binary number can be found by expanding the number into a power series with a base of 2. For example,

$$(11010)_2 = 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = (26)_{10}$$

As noted earlier, the digits in a binary number are called bits. When a bit is equal to 0, it does not contribute to the sum during the conversion. Therefore, the conversion to decimal can be obtained by adding the numbers with powers of two corresponding to the bits that are equal to 1. For example,

$$(110101.11)_2 = 32 + 16 + 4 + 1 + 0.5 + 0.25 = (53.75)_{10}$$

The first 24 numbers obtained from 2 to the power of n are listed in Table 1-1. In computer work, 2^{10} is referred to as K (kilo), 2^{20} as M (mega), and 2^{30} as G (giga). Thus,

$$4\text{K} = 2^2 \times 2^{10} = 2^{12} = 4,096 \text{ and } 16\text{M} = 2^4 \times 2^{20} = 2^{24} = 16,777,216$$

The conversion of a decimal number to binary can be easily achieved by a method that successively subtracts powers of two from the decimal number. To convert the decimal number N to binary, first find the greatest number that is a power of two (see Table 1-1) and that, subtracted from N , produces a positive difference. Let the difference be designated N_1 . Now find the greatest number that is a power of two and that, subtracted from N_1 , produces a positive difference N_2 . Continue this procedure until the difference is zero. In this way, the decimal number is converted to its powers-of-two components. The equivalent binary number is obtained from the coefficients of a power series that forms the sum of the components. 1's appear in the binary number in the positions for which terms appear in the power series, and 0's appear in all other positions. This method is demonstrated by the conversion of decimal 625 to binary as follows:

$$\begin{aligned} 625 - 512 &= 113 = N_1 & 512 &= 2^9 \\ 113 - 64 &= 49 = N_2 & 64 &= 2^6 \end{aligned}$$

$$\begin{aligned}
 49 - 32 &= 17 = N_3 & 32 &= 2^5 \\
 17 - 16 &= 1 = N_4 & 16 &= 2^4 \\
 1 - 1 &= 0 = N_5 & 1 &= 2^0 \\
 (625)_{10} &= 2^9 + 2^6 + 2^5 + 2^4 + 2^0 = (1001110001)_2
 \end{aligned}$$

Octal and Hexadecimal Numbers

As previously mentioned, all computers and digital systems use the binary representation. The octal (base-8) and hexadecimal (base-16) systems are useful for representing binary quantities indirectly because they possess the property that their bases are powers of two. Since $2^3 = 8$ and $2^4 = 16$, each octal digit corresponds to three binary digits and each hexadecimal digit corresponds to four binary digits.

The more compact representation of binary numbers in either octal or hexadecimal is much more convenient for people than using bit strings in binary that are three to four times as long. Thus, most computer manuals use either octal or hexadecimal numbers to specify binary quantities. A group of 15 bits, for example, can be represented in the octal system with only five digits. A group of 16 bits can be represented in hexadecimal with four digits. The choice between an octal and a hexadecimal representation of binary numbers is arbitrary, although hexadecimal tends to win out, since bits often appear in a group of size divisible by four.

The octal number system is the base-8 system with digits 0, 1, 2, 3, 4, 5, 6, 7. An example of an octal number is 127.4. To determine its equivalent decimal value, we expand the number in a power series with a base of 8:

$$(127.4)_8 = 1 \times 8^2 + 2 \times 8^1 + 7 \times 8^0 + 4 \times 8^{-1} = (87.5)_{10}$$

Note that the digits 8 and 9 cannot appear in an octal number.

It is customary to use the first r digits from the decimal system, starting with 0, to represent the coefficients in a base- r system when r is less than 10. The letters of the alphabet are used to supplement the digits when r is 10 or more. The hexadecimal number system is a base-16 system with the first 10 digits borrowed from the decimal system and the letters A, B, C, D, E, and F used for the values 10, 11, 12, 13, 14, and 15, respectively. An example of a hexadecimal number is

$$(B65F)_{16} = 11 \times 16^3 + 6 \times 16^2 + 5 \times 16^1 + 15 \times 16^0 = (46687)_{10}$$

The first 16 numbers in the decimal, binary, octal, and hexadecimal number systems are listed in Table 1-2. Note that the sequence of binary numbers follows a prescribed pattern. The least significant bit alternates between 0 and 1, the second significant bit alternates between two 0's and two 1's, the third significant bit alternates between four 0's and four 1's, and the most significant bit alternates between eight 0's and eight 1's.

The conversion from binary to octal is easily accomplished by partitioning the binary number into groups of three bits each, starting from the binary point

□ TABLE 1-2
Numbers with Different Bases

Decimal (base 10)	Binary (base 2)	Octal (base 8)	Hexadecimal (base 16)
00	0000	00	0
01	0001	01	1
02	0010	02	2
03	0011	03	3
04	0100	04	4
05	0101	05	5
06	0110	06	6
07	0111	07	7
08	1000	10	8
09	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

and proceeding to the left and to the right. The corresponding octal digit is then assigned to each group. The following example illustrates the procedure:

$$(010\ 110\ 001\ 101\ 011.\ 111\ 100\ 000\ 110)_2 = (26153.7406)_8$$

The corresponding octal digit for each group of three bits is obtained from the first eight entries in Table 1-2. To make the total count of bits a multiple of three, 0's can be added on the left of the string of bits to the left of the binary point. More importantly, 0's must be added on the right of the string of bits to the right of the binary point to make the number of bits a multiple of three and obtain the correct octal result.

Conversion from binary to hexadecimal is similar, except that the binary number is divided into groups of four digits. The previous binary number is converted to hexadecimal as follows:

$$(0010\ 1100\ 0110\ 1011.\ 1111\ 0000\ 0110)_2 = (2C6B.F06)_{16}$$

The corresponding hexadecimal digit for each group of four bits is obtained by reference to Table 1-2.

Conversion from octal or hexadecimal to binary is done by reversing the procedure just performed. Each octal digit is converted to a 3-bit binary equivalent and extra 0's are deleted. Similarly, each hexadecimal digit is converted to its 4-bit binary equivalent. This is illustrated in the following examples:

$$\begin{aligned}(673.12)_8 &= 110\ 111\ 011.\ 001\ 010 = (110111011.00101)_2 \\ (3A6.C)_{16} &= 0011\ 1010\ 0110.\ 1100 = (1110100110.11)_2\end{aligned}$$

Number Ranges

In digital computers, the range of numbers that can be represented is based on the number of bits available in the hardware structures that store and process information. The number of bits in these structures is most frequently a power of two, such as 8, 16, 32, and 64. Since the numbers of bits is fixed by the structures, the addition of leading or trailing zeros to represent numbers is necessary, and the range of numbers that can be represented is also fixed.

For example, for a computer processing 16-bit unsigned integers, the number 537 is represented as 0000001000011001. The range of integers that can be handled by this representation is from 0 to $2^{16} - 1$, that is, from 0 to 65,535. If the same computer is processing 16-bit unsigned fractions with the binary point to the left of the most significant digit, then the number 0.375 is represented by 0.0110000000000000. The range of fractions that can be represented is from 0 to $(2^{16} - 1)/2^{16}$, or from 0.0 to 0.9999847412.

In later chapters, we will deal with fixed-bit representations and ranges for binary signed numbers and floating-point numbers. In both of these cases, some bits are used to represent information other than simple integer or fraction values.

1-3 ARITHMETIC OPERATIONS

Arithmetic operations with numbers in base r follow the same rules as for decimal numbers. However, when a base other than the familiar base 10 is used, one must be careful to use only r allowable digits and perform all computations with base- r digits. Examples of the addition of two binary numbers are as follows (note the names of the operands for addition):

Carries:	00000	101100
Augend:	01100	10110
Addend:	+10001	+10111
Sum:	<u>11101</u>	<u>101101</u>

The sum of two binary numbers is calculated following the same rules as for decimal numbers, except that the sum digit in any position can be only 1 or 0. Also, a carry in binary occurs if the sum in any bit position is greater than 1. (A carry in decimal occurs if the sum in any digit position is greater than 9.) Any carry obtained in a given position is added to the bits in the column one significant position higher. In the first example, since all of the carries are 0, the sum bits are simply the sum of the augend and addend bits. In the second example, the sum of the bits in the second column from the right is 2, giving a sum bit of 0 and a carry bit of 1 ($2 = 2 + 0$). The carry bit is added with the 1's in the third position, giving a sum of 3, which produces a sum bit of 1 and a carry of 1 ($3 = 2 + 1$).

The following are examples of the subtraction of two binary numbers; as with addition, note the names of the operands:

Borrows:	00000	00110		00110
Minuend:	10110	10110	10011	11110
Subtrahend:	-10010	-10011	-11110	-10011
Difference:	<u>00100</u>	<u>00011</u>		<u>-01011</u>

The rules for subtraction are the same as in decimal, except that a borrow into a given column adds 2 to the minuend bit. (A borrow in the decimal system adds 10 to the minuend digit.) In the first example shown, no borrows occur, so the difference bits are simply the minuend bits minus the subtrahend bits. In the second example, in the right position, the subtrahend bit is 1 with the minuend bit 0, so it is necessary to borrow from the second position as shown. This gives a difference bit in the first position of 1 ($2 + 0 - 1 = 1$). In the second position, the borrow is subtracted, so a borrow is again necessary. Recall that, in the event that the subtrahend is larger than the minuend, we subtract the minuend from the subtrahend and give the result a minus sign. This is the case in the third example, in which this interchange of the two operands is shown.

The final operation to be illustrated is binary multiplication, which is quite simple. The multiplier digits are always 1 or 0. Therefore, the partial products are equal either to the multiplicand or to 0. Multiplication is illustrated by the following example:

Multiplicand:	1011
Multiplier:	$\times 101$
	<u>1011</u>
	0000
	1011
Product:	<u>110111</u>

Arithmetic operations with octal, hexadecimal, or any other base- r system will normally require the formulation of tables from which one obtains sums and products of two digits in that base. An easier alternative for adding two numbers in base r is to convert each pair of digits in a column to decimal, add the digits in decimal, and then convert the result to the corresponding sum and carry in the base- r system. Since addition is done in decimal, we can rely on our memories for obtaining the entries from the familiar decimal addition table. The sequence of steps for adding the two hexadecimal numbers 59F and E46 is shown in Example 1-1.

EXAMPLE 1-1 Hexadecimal AdditionPerform the addition $(59F)_{16} + (E46)_{16}$:

Hexadecimal	Equivalent Decimal Calculation	
$\begin{array}{r} 59F \\ E46 \\ \hline 13E5 \end{array}$	$\begin{array}{r} 1 \leftarrow \\ 5 \\ 14 \\ \hline 19 = 16 + 3 \end{array}$	$\begin{array}{r} 1 \leftarrow \\ 9 \quad 15 \\ 4 \quad 6 \\ \hline 14 = E \quad 21 = 16 + 5 \end{array}$

The equivalent decimal calculation columns on the right show the mental reasoning that must be carried out to produce each digit of the hexadecimal sum. Instead of adding $F + 6$ in hexadecimal, we add the equivalent decimals, $15 + 6 = 21$. We then convert back to hexadecimal by noting that $21 = 16 + 5$. This gives a sum digit of 5 and a carry of 1 to the next higher order column of digits. The other two columns are added in a similar fashion. ■

The multiplication of two base- r numbers can be accomplished by doing all the arithmetic operations in decimal and converting intermediate results one at a time. This is illustrated in the multiplication of two octal numbers shown next in Example 1-2.

EXAMPLE 1-2 Octal MultiplicationPerform the multiplication $(762)_8 \times (45)_8$:

Octal	Octal	Decimal	Octal
762	5×2	$= 10 = 8 + 2$	$= 12$
45	$5 \times 6 + 1$	$= 31 = 24 + 7$	$= 37$
4672	$5 \times 7 + 3$	$= 38 = 32 + 6$	$= 46$
3710	4×2	$= 8 = 8 + 0$	$= 10$
43772	$4 \times 6 + 1$	$= 25 = 24 + 1$	$= 31$
	$4 \times 7 + 3$	$= 31 = 24 + 7$	$= 37$

The computations on the right show the mental calculations for each pair of octal digits. The octal digits 0 through 7 have the same value as their corresponding decimal digits. The multiplication of two octal digits plus a carry, derived from the calculation on the previous line, is done in decimal, and the result is then converted back to octal. The left digit of the two-digit octal result gives the carry that must be added to the digit product on the next line. The blue digits from the octal results of the decimal calculations are copied to the octal partial products on the left. For example, $(5 \times 2)_8 = (12)_8$. The left digit, 1, is the carry to be added to the product $(5 \times 6)_8$, and the blue least significant digit, 2, is the corresponding digit of the octal partial product. When there is no digit product to which the carry can be added, the carry is written directly into the octal partial product, as in the case of the 4 in 46. ■

Conversion from Decimal to Other Bases

The conversion of a number in base r to decimal is done by expanding the number in a power series and adding all the terms, as shown previously. We now present a general procedure for the reverse operation of converting a decimal number to a number in base r that is related to the alternative expansion to decimal in Section 1-2. If the number includes a radix point, it is necessary to separate the number into an integer part and a fraction part, since the two parts must be converted differently. The conversion of a decimal integer to a number in base r is done by dividing the number and all successive quotients by r and accumulating the remainders. This procedure is best explained by example.

EXAMPLE 1-3 Conversion of Decimal Integers to Octal

Convert decimal 153 to octal:

The conversion is to base 8. First, 153 is divided by 8 to give a quotient of 19 and a remainder of 1, as shown in blue. Then 19 is divided by 8 to give a quotient of 2 and a remainder of 3. Finally, 2 is divided by 8 to give a quotient of 0 and a remainder of 2. The coefficients of the desired octal number are obtained from the remainders:

$153/8 = 19 + 1/8$	Remainder = 1	↑	Least significant digit
$19/8 = 2 + 3/8$	= 3		
$2/8 = 0 + 2/8$	= 2		Most significant digit

$(153)_{10} = (231)_8$ ■

Note in Example 1-3 that the remainders are read from last to first, as indicated by the arrow, to obtain the converted number. The quotients are divided by r until the result is 0. We also can use this procedure to convert decimal integers to binary as shown in Example 1-4. In this case, the base of the converted number is 2, and therefore, all the divisions must be done by 2.

EXAMPLE 1-4 Conversion of Decimal Integers to Binary

Convert decimal 41 to binary:

$41/2 = 20 + 1/2$	Remainder = 1	↑	Least significant digit
$20/2 = 10$	= 0		
$10/2 = 5$	= 0		
$5/2 = 2 + 1/2$	= 1		
$2/2 = 1$	= 0		
$1/2 = 0 + 1/2$	= 1		Most significant digit

$(41)_{10} = (101001)_2$

Of course, the decimal number could be converted by the sum of powers of two:

$$(41)_{10} = 32 + 8 + 1 = (101001)_2 \quad \blacksquare$$

The conversion of a decimal fraction to base r is accomplished by a method similar to that used for integers, except that multiplication by r is used instead of division, and integers are accumulated instead of remainders. Again, the method is best explained by example.

EXAMPLE 1-5 Conversion of Decimal Fractions to Binary

Convert decimal 0.6875 to binary:

First, 0.6875 is multiplied by 2 to give an integer and a fraction. The new fraction is multiplied by 2 to give a new integer and a new fraction. This process is continued until the fractional part equals 0 or until there are enough digits to give sufficient accuracy. The coefficients of the binary number are obtained from the integers as follows:

$0.6875 \times 2 = 1.3750$	Integer = 1	Most significant digit
$0.3750 \times 2 = 0.7500$	= 0	<div style="display: flex; align-items: center; justify-content: center;"> <div style="margin-right: 5px;">↓</div> <div style="margin-right: 5px;">Least significant digit</div> </div>
$0.7500 \times 2 = 1.5000$	= 1	
$0.5000 \times 2 = 1.0000$	= 1	
$(0.6875)_{10} = (0.1011)_2$		

Note in the foregoing example that the integers are read from first to last, as indicated by the arrow, to obtain the converted number. In the example, a finite number of digits appears in the converted number. The process of multiplying fractions by r does not necessarily end with zero, so we must decide how many digits of the fraction to use from the conversion. Also, remember that the multiplications are by number r . Therefore, to convert a decimal fraction to octal, we must multiply the fractions by 8, as shown in Example 1-6.

EXAMPLE 1-6 Conversion of Decimal Fractions to Octal

Convert decimal 0.513 to a three-digit octal fraction:

$0.513 \times 8 = 4.104$	Integer = 4	Most significant digit
$0.104 \times 8 = 0.832$	= 0	<div style="display: flex; align-items: center; justify-content: center;"> <div style="margin-right: 5px;">↓</div> <div style="margin-right: 5px;">Least significant digit</div> </div>
$0.832 \times 8 = 6.656$	= 6	
$0.656 \times 8 = 5.248$	= 5	

The answer, to three significant figures, is obtained from the integer digits. Note that the last integer digit, 5, is used for rounding in base 8 of the second-to-the-last digit, 6, to obtain

$$(0.513)_{10} = (0.407)_8 \quad \blacksquare$$

The conversion of decimal numbers with both integer and fractional parts is done by converting each part separately and then combining the two answers. Using the results of Example 1-3 and Example 1-6, we obtain

$$(153.513)_{10} = (231.407)_8$$

1-4 DECIMAL CODES

The binary number system is the most natural system for a computer, but people are accustomed to the decimal system. One way to resolve this difference is to convert decimal numbers to binary, perform all arithmetic calculations in binary, and then convert the binary results back to decimal. This method requires that we store the decimal numbers in the computer in a way that they can be converted to binary. Since the computer can accept only binary values, we must represent the decimal digits by a code that contains 1's and 0's. It is also possible to perform the arithmetic operations directly with decimal numbers when they are stored in the computer in coded form.

An n -bit *binary code* is a group of n bits that assume up to 2^n distinct combinations of 1's and 0's, with each combination representing one element of the set being coded. A set of four elements can be coded with a 2-bit binary code, with each element assigned one of the following bit combinations: 00, 01, 10, 11. A set of 8 elements requires a 3-bit code, and a set of 16 elements requires a 4-bit code. The bit combinations of an n -bit code can be determined from the count in binary from 0 to $2^n - 1$. Each element must be assigned a unique binary bit combination, and no two elements can have the same value; otherwise, the code assignment is ambiguous.

A binary code will have some unassigned bit combinations if the number of elements in the set is not a power of 2. The 10 decimal digits form such a set. A binary code that distinguishes among 10 elements must contain at least four bits, but six out of the 16 possible combinations will remain unassigned. Numerous different binary codes can be obtained by arranging four bits into 10 distinct combinations. The code most commonly used for the decimal digits is the straightforward binary assignment listed in Table 1-2 on page 12. This is called *binary-coded decimal* and is commonly referred to as BCD. Other decimal codes are possible, a few of which are presented in Chapter 3.

Table 1-3 gives a 4-bit code for each decimal digit. A number with n decimal digits will require $4n$ bits in BCD. Thus, decimal 396 is represented in BCD with 12 bits as

$$0011 \ 1001 \ 0110$$

with each group of four bits representing one decimal digit. A decimal number in BCD is the same as its equivalent binary number only when the number is between 0 and 9, inclusive. A BCD number greater than 10 has a representation different from its equivalent binary number, even though both contain 1's and 0's.

□ **TABLE 1-3**
Binary-Coded Decimal (BCD)

Decimal Symbol	BCD Digit
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Moreover, the binary combinations 1010 through 1111 are not used and have no meaning in the BCD code.

Consider decimal 185 and its corresponding value in BCD and binary:

$$(185)_{10} = (0001\ 1000\ 0101)_{\text{BCD}} = (10111001)_2$$

The BCD value has 12 bits, but the equivalent binary number needs only 8 bits. It is obvious that a BCD number needs more bits than its equivalent binary value. However, there is an advantage in the use of decimal numbers because computer input and output data are handled by people who use the decimal system. BCD numbers are decimal numbers and not binary numbers, even though they are represented in bits. The only difference between a decimal and a BCD number is that decimals are written with the symbols 0, 1, 2, ..., 9, and BCD numbers use the binary codes 0000, 0001, 0010, ..., 1001.

BCD Addition

Consider the addition of two decimal digits in BCD, together with a possible carry of 1 from a previous less significant pair of digits. Since each digit does not exceed 9, the sum cannot be greater than $9 + 9 + 1 = 19$, the 1 being a carry. Suppose we add the BCD digits as if they were binary numbers. Then the binary sum will produce a result in the range from 0 to 19. In binary, this will be from 0000 to 10011, but in BCD, it should be from 0000 to 1 1001, the first 1 being a carry and the next four bits being the BCD digit sum. When the binary sum is less than 1010 (without a carry), the corresponding BCD digit is correct. But when the binary sum is greater than or equal to 1010, the result is an invalid BCD digit. The addition of binary 6, $(0110)_2$, to the sum converts it to the correct digit and also produces a decimal carry as required. The reason is that the difference between a carry from the most significant bit position of the binary sum and a decimal carry is $16 - 10 = 6$.

Thus, the decimal carry and the correct BCD sum digit are forced by adding 6 in binary. Consider the next three-digit BCD addition example.

EXAMPLE 1-7 BCD Addition

110	BCD carry			
448		0100	0100	1000
+489		+0100	+1000	+1001
<hr/> 937	Binary sum	<hr/> 1001	<hr/> 1101	<hr/> 1 0001
	Add 6		+0110	+0110
	BCD sum		<hr/> 1 0011	<hr/> 1 0111
	BCD result	1001	0011	0111

In each position, the two BCD digits are added as if they were two binary numbers. If the binary sum is greater than 1001, we add 0110 to obtain the correct BCD digit sum and a carry. In the right column, the binary sum is equal to 17. The presence of the carry indicates that the sum is greater than 16 (certainly greater than 9), so a correction is needed. The addition of 0110 produces the correct BCD digit sum, 0111 (7), and a carry of 1. In the next column, the binary sum is 1101 (13), an invalid BCD digit. Addition of 0110 produces the correct BCD digit sum, 0011 (3), and a carry of 1. In the final column, the binary sum is equal to 1001 (9) and is the correct BCD digit. ■

Parity Bit

To detect errors in data communication and processing, an additional bit is sometimes added to a binary code word to define its parity. A *parity bit* is the extra bit included to make the total number of 1's in the resulting code word either even or odd. Consider the following two characters and their even and odd parity:

	With Even Parity	With Odd Parity
1000001	01000001	11000001
1010100	11010100	01010100

In each case, we use the extra bit in the leftmost position of the code to produce an even number of 1's in the character for even parity or an odd number of 1's in the character for odd parity. In general, one parity or the other is adopted, with even parity being more common. Parity may be used with binary numbers as well as with codes, including ASCII for characters, and the parity bit may be placed in any fixed position in the code.

The parity bit is helpful in detecting errors during the transmission of information from one location to another. Assuming that even parity is used, the simplest case is handled as follows: An even (or odd) parity bit is generated at the sending end for all 7-bit ASCII characters; the 8-bit characters that include parity bits are transmitted to their destination. The parity of each character is then checked at the receiving end; if the parity of the received character is not even (odd), it means that at least one bit has changed its value during the transmission. This method detects one, three, or any odd number of errors in each character transmitted. An even number of errors is undetected. Other error-detection codes, some of which are based on additional parity bits, may be needed to take care of an even number of errors. What is done after an error is detected depends on the particular application. One possibility is to request retransmission of the message on the assumption that the error was random and will not occur again. Thus, if the receiver detects a parity error, it sends back a NAK (negative acknowledge) control character consisting of the even-parity eight bits, 10010101, from Table 1-5 on page 25. If no error is detected, the receiver sends back an ACK (acknowledge) control character, 00000110. The sending end will respond to a NAK by transmitting the message again, until the correct parity is received. If, after a number of attempts, the transmission is still in error, an indication of a malfunction in the transmission path is given.

1-5 GRAY CODES

As we count up or down using binary codes, the number of bits that change from one binary value to the next varies. This is illustrated by the binary code for the octal digits on the left in Table 1-4. As we count from 000 up to 111 and “roll over” to 000, the number of bits that change between the binary values ranges from 1 to 3.

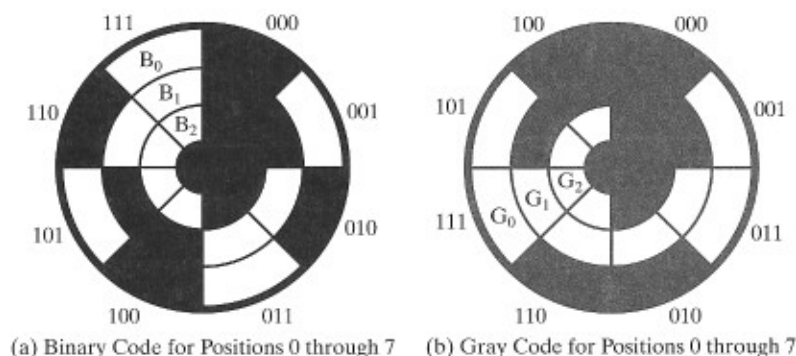
□ TABLE 1-4
Gray Code

Binary Code	Bit Changes	Gray Code	Bit Changes
000	1	000	1
001	2	001	1
010	1	011	1
011	3	010	1
100	1	110	1
101	2	111	1
110	1	101	1
111	3	100	1
000		000	1

For many applications, multiple bit changes as the circuit counts is not a problem. There are applications, however, in which a change of more than one bit when counting up or down can cause serious problems. One such problem is illustrated by an optical shaft angle encoder shown in Figure 1-3(a). The encoder is a disk attached to a rotating shaft for measurement of the rotational position of the shaft. The disk contains areas that are clear for binary 1 and opaque for binary 0. An illumination source is placed on one side of the disk, and optical sensors, one for each of the bits to be encoded, are placed on the other side of the disk. When a clear region lies between the source and a sensor, the sensor responds to the light with a binary 1 output. When an opaque region lies between the source and the sensor, the sensor responds to the dark with a binary 0.

The rotating shaft, however, can be in any angular position. For example, suppose that the shaft and disk are positioned so that the sensors lie right at the boundary between 011 and 100. In this case, sensors in positions B_2 , B_1 and B_0 have the light partially blocked. In such a situation, it is unclear whether the three sensors will see light or dark. As a consequence, each sensor may produce either a 1 or a 0. Thus, the resulting encoded binary number for a value between 3 and 4 may be 000, 001, 010, 011, 100, 101, 110, or 111. Either 011 or 100 will be satisfactory in this case, but the other six values are clearly erroneous!

The solution to this problem becomes apparent by noting that in those cases in which only a single bit changes when going from one value to the next or previous value, this problem cannot occur. For example, if the sensors lie on the boundary between 2 and 3, the resulting code is either 010 or 011, either of which is satisfactory. If we change the encoding of the values 0 through 7 such that only one bit value changes as we count up or down (including rollover from 7 to 0), then the encoding will be satisfactory for all positions. A code having the property that only one bit at a time changes between codes during counting is a *Gray code*. There are multiple Gray codes for any set of n consecutive integers, with n even.



□ **FIGURE 1-3**
Optical Shaft Angle Encoder

A specific Gray code for the octal digits, called a *binary reflected Gray code*, appears on the right in Table 1-4. Note that the counting order for binary codes is now 000, 001, 011, 010, 110, 111, 101, 100, and 000. If we want binary codes for processing, then we can build a digital circuit or use software that converts these codes to binary before they are used in further processing of the information.

Figure 1-3(b) shows the optical shaft angle encoder using the Gray code from Table 1-4. Note that any two segments on the disk adjacent to each other have only one region that is clear for one and opaque for the other. The Gray code is named for Frank Gray who patented its use for shaft encoders in 1953.

The optical shaft encoder illustrates one use of the Gray code concept. There are many other similar uses in which a physical variable, such as position or voltage, has a continuous range of values that is converted to a digital representation. A quite different use of Gray codes appears in low-power CMOS (Complementary Metal Oxide Semiconductor) logic circuits that count up or down. In CMOS, power is consumed only when a bit changes. For the example codes given in Table 1-4 with continuous counting (either up or down), there are 14 bit changes for binary counting for every eight bit changes for Gray code counting. Thus, the power consumed at the counter outputs for the Gray code counter is only 57% of that consumed at the binary counter outputs.

A Gray code for a counting sequence of n binary code words (n must be even) can be constructed by replacing each of the first $n/2$ numbers in the sequence with a code word consisting of 0 followed by the even parity for each bit of the binary code word and the bit to its left. For example, for the binary code word 0100, the Gray code word is 0, parity(0,1), parity(1,0), parity(0,0) = 0110. Next, take the sequence of numbers formed and copy it in reverse order with the leftmost 0 replaced by a 1. This new sequence provides the Gray code words for the second $n/2$ of the original n code words. For example, for BCD codes, the first five Gray code words are 0000, 0001, 0011, 0010, and 0110. Reversing the order of these codes and replacing the leftmost 0 with a 1, we obtain 1110, 1010, 1011, 1001, and 1000 for the last five Gray codes.

For the special cases in which the original binary codes are 0 through $2^n - 1$, each Gray code word may be formed directly from the corresponding binary code word by copying its leftmost bit and then replacing each of the remaining bits with the even parity of the bit of the number and the bit to its left.

1-6 ALPHANUMERIC CODES

Many applications of digital computers require the handling of data consisting not only of numbers, but also of letters. For instance, an insurance company with thousands of policyholders uses a computer to process its files. To represent the names and other pertinent information, it is necessary to formulate a binary code for the letters of the alphabet. In addition, the same binary code must represent numerals and special characters such as \$. Any alphanumeric character set for English is a set of elements that includes the 10 decimal digits, the 26 letters of the alphabet, and several (more than three) special characters. If only capital letters are

included, we need a binary code of at least six bits, and if both uppercase letters and lowercase letters are included, we need a binary code of at least seven bits. Binary codes play an important role in digital computers. The codes must be in binary because computers can handle only 1's and 0's. Note that binary encoding merely changes the symbols, not the meaning of the elements of information being encoded.

ASCII Character Code

The standard binary code for the alphanumeric characters is called ASCII (American Standard Code for Information Interchange). It uses seven bits to code 128 characters, as shown in Table 1-5. The seven bits of the code are designated by B_1 through B_7 , with B_7 being the most significant bit. Note that the most significant three bits of the code determine the column of the table and the least significant four bits the row of the table. The letter A, for example, is represented in ASCII as 1000001 (column 100, row 0001). The ASCII code contains 94 characters that can be printed and 34 nonprinting characters used for various control functions. The printing characters consist of the 26 uppercase letters, the 26 lowercase letters, the 10 numerals, and 32 special printable characters such as %, @, and \$.

The 34 control characters are designated in the ASCII table with abbreviated names. They are listed again below the table with their full functional names. The control characters are used for routing data and arranging the printed text into a prescribed format. There are three types of control characters: format effectors, information separators, and communication control characters. Format effectors are characters that control the layout of printing. They include the familiar typewriter controls such as backspace (BS), horizontal tabulation (HT), and carriage return (CR). Information separators are used to separate the data into divisions—for example, paragraphs and pages. They include characters such as record separator (RS) and file separator (FS). The communication control characters are used during the transmission of text from one location to the other. Examples of communication control characters are STX (start of text) and ETX (end of text), which are used to frame a text message transmitted via communication wires.

ASCII is a 7-bit code, but most computers manipulate an 8-bit quantity as a single unit called a *byte*. Therefore, ASCII characters most often are stored one per byte, with the most significant bit set to 0. The extra bit is sometimes used for specific purposes, depending on the application. For example, some printers recognize an additional 128 8-bit characters, with the most significant bit set to 1. These characters enable the printer to produce additional symbols, such as those from the Greek alphabet or characters with accent marks as used in languages other than English.



UNICODE This supplement on Unicode, a 16-bit standard code for representing the symbols and ideographs for the world's languages, is available on the Companion Website (<http://www.prenhall.com/mano>) for the text.

□ **TABLE 1-5**
American Standard Code for Information Interchange (ASCII)

$B_4B_3B_2B_1$	$B_7B_6B_5$							
	000	001	010	011	100	101	110	111
0000	NULL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	\$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(8	H	X	h	x
1001	HT	EM)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[k	{
1100	FF	FS	,	<	L	\	l	
1101	CR	GS	-	=	M]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

Control Characters:

NULL	NULL	DLE	Data link escape
SOH	Start of heading	DC1	Device control 1
STX	Start of text	DC2	Device control 2
ETX	End of text	DC3	Device control 3
EOT	End of transmission	DC4	Device control 4
ENQ	Enquiry	NAK	Negative acknowledge
ACK	Acknowledge	SYN	Synchronous idle
BEL	Bell	ETB	End of transmission block
BS	Backspace	CAN	Cancel
HT	Horizontal tab	EM	End of medium
LF	Line feed	SUB	Substitute
VT	Vertical tab	ESC	Escape
FF	Form feed	FS	File separator
CR	Carriage return	GS	Group separator
SO	Shift out	RS	Record separator
SI	Shift in	US	Unit separator
SP	Space	DEL	Delete

1-7 CHAPTER SUMMARY

In this chapter, we have introduced digital systems and digital computers and have shown why such systems use signals having only two values. We have briefly introduced computer structure with a block diagram and discussion of the nature of the

blocks. Number system concepts, including base (radix) and radix point, were presented. Because of their correspondence to two-valued signals, binary numbers were discussed in detail. Octal (base 8) and hexadecimal (base 16) were also emphasized, since they are useful as shorthand notation for binary. Arithmetic operations in bases other than base 10 and the conversion of numbers from one base to another were covered. Because of the predominance of decimal in normal use, Binary Coded Decimal (BCD) was treated. The parity bit was presented as a technique for error detection, and the Gray code, which is critical to selected applications, was defined. Finally, the representation of information in the form of characters instead of numbers by means of the ASCII code for the English alphabet was presented.

In subsequent chapters, we will treat the representation of signed numbers and floating-point numbers. We will also introduce additional codes for the decimal digits. Although these topics fit well with the topics in this chapter, they are difficult to motivate without associating them with the hardware used to implement the operations they denote. Thus, we delay their presentation until we examine the associated hardware.

REFERENCES

1. GRAY, F. *Pulse Code Communication*. U. S. Patent 2 632 058, March 17, 1953.
2. MANO, M. M. *Computer Engineering: Hardware Design*. Englewood Cliffs, NJ: Prentice Hall, 1988.
3. MANO, M. M. *Digital Design*, 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 2002.
4. MANO, M. M. *Computer System Architecture*, 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1993.
5. PATTERSON, D. A., AND HENNESSY, J. L. *Computer Organization and Design: The Hardware/Software Interface*. 2nd ed. San Mateo, CA: Morgan Kaufmann, 1998.
6. TANENBAUM, A. S. *Structured Computer Organization*, 4th ed. Upper Saddle River, NJ: Prentice Hall, 1999.
7. WHITE, R. *How Computers Work*, Emeryville, CA: Ziff-Davis Press, 1993.
8. WILLIAMS, M. R. *A History of Computing Technology*. Englewood Cliffs, NJ: Prentice-Hall, 1985.

PROBLEMS



The plus (+) indicates a more advanced problem and the asterisk (*) indicates a solution is available on the Companion Website for the text.

- 1-1. *List the binary, octal, and hexadecimal numbers from 16 to 31.
- 1-2. What is the exact number of bits in a memory that contains (a) 48K bits; (b) 384M bits; (c) 8G bits?

- 1-3. What is the decimal equivalent of the largest binary integer that can be obtained with (a) 12 bits and (b) 24 bits?
- 1-4. *Convert the following binary numbers to decimal: 1001101, 1010011.101, and 10101110.1001.
- 1-5. Convert the following decimal numbers to binary: 125, 610, 2003, and 18944.
- 1-6. Each of the following five numbers has a different base: $(11100111)_2$, $(22120)_3$, $(3113)_4$, $(4110)_5$, and $(343)_8$. Which of the five numbers have the same value in decimal?
- 1-7. *Convert the following numbers from the given base to the other three bases listed in the table:

Decimal	Binary	Octal	Hexadecimal
369.3125	?	?	?
?	10111101.101	?	?
?	?	326.5	?
?	?	?	F3C7.A

- 1-8. *Convert the following decimal numbers to the indicated bases using the methods of Examples 1-3 on page 16 and 1-6 on page 17:
(a) 7562.45 to octal (b) 1938.257 to hexadecimal (c) 175.175 to binary.
- 1-9. *Perform the following conversion by using base 2 instead of base 10 as the intermediate base for the conversion:
(a) $(673.6)_8$ to hexadecimal (b) $(E7C.B)_{16}$ to octal (c) $(310.2)_4$ to octal
- 1-10. Perform the following binary multiplications:
(a) 1101×1001 (b) 0101×1011 (c) 100101×0110110
- 1-11. +Division is composed of multiplications and subtractions. Perform the binary division $1011110 \div 101$ to obtain a quotient and remainder.
- 1-12. There is considerable evidence to suggest that base 20 has historically been used for number systems in a number of cultures.
(a) Write the digits for a base-20 system, using an extension of the same digit representation scheme employed for hexadecimal.
(b) Convert $(2003)_{10}$ to base 20. (c) Convert $(BCH.G)_{20}$ to decimal.
- 1-13. *In each of the following cases, determine the radix r :
(a) $(BEE)_r = (2699)_{10}$ (b) $(365)_r = (194)_{10}$
- 1-14. The following calculation was performed by a particular breed of unusually intelligent chicken. If the radix r used by the chicken corresponds to its total number of toes, how many toes does the chicken have on each foot?
$$((35)_r + (24)_r) \times (21)_r = (1501)_r$$
- 1-15. *Represent the decimal numbers 694 and 835 in BCD, and then show the steps necessary to form their sum.

- 1-16.** *Find the binary representations for each of the following BCD numbers:
(a) 0100 1000 0110 0111 (b) 0011 0111 1000.0111 0101
- 1-17.** List the 5-bit binary number equivalents for 16 through 31 with a parity bit added in the rightmost position giving odd parity to the overall 6-bit numbers. Repeat for even parity.
- 1-18.** By using the procedure given in Section 1-5, find the Gray code for the hexadecimal digits.
- 1-19.** + What is the percentage of power consumed for continuous counting (either up or down but not both) at the outputs of a binary Gray code counter compared to a binary counter as a function of the number of bits, n , in the two counters?
- 1-20.** What bit position in an ASCII code must be complemented to change the ASCII letter represented from uppercase to lowercase and vice versa?
- 1-21.** Write your full name in ASCII, using an 8-bit code (a) with the leftmost bit always 0 and (b) with the leftmost bit selected to produce even parity. Include a space between names and a period after the middle initial.
- 1-22.** Decode the following ASCII code: 1001010 1101111 1101000 1101110 0100000 1000100 1101111 1100101.
- 1-23.** *Show the bit configuration that represents the decimal number 365 in (a) binary, (b) BCD, (c) ASCII.
- 1-24.** A computer represents information in groups of 32 bits. How many different integers can be represented in (a) binary, (b) BCD, and (c) 8-bit ASCII, all using 32 bits?

CHAPTER

2

COMBINATIONAL LOGIC CIRCUITS

In this chapter, we will learn about gates, the most primitive logic elements used in digital systems. In addition, we will learn the mathematical techniques used in designing circuits from these gates and learn how to design cost-effective circuits. These techniques, which are fundamental to the design of almost all digital circuits, are based on Boolean algebra. One aspect of design is to avoid unnecessary circuitry and excess cost, a goal accomplished by a technique called optimization. Karnaugh maps provide a graphical method for enhancing understanding of optimization and solving small optimization problems for "two-level" logic circuits. More general optimization methods for circuits with more than two levels are introduced. Types of logic gates characteristic of contemporary integrated circuit implementation are discussed. Exclusive OR and Exclusive NOR gates are introduced, along with associated algebraic techniques.

In terms of the diagram at the beginning of Chapter 1, concepts from this chapter apply to most of the generic computer. Exceptions are circuits that are largely memory, such as caches and RAM, and analog electronic circuits in the monitor and hard disk controller. Nevertheless, with its use throughout the design of most of the computer, what we study in this chapter is fundamental to an in-depth understanding of computers and digital systems and how they are designed.

2-1 BINARY LOGIC AND GATES

Digital circuits are hardware components that manipulate binary information. The circuits are implemented using transistors and interconnections in complex semiconductor devices called *integrated circuits*. Each basic circuit is referred to as a *logic gate*. For simplicity in design, we model the transistor-based electronic

circuits as logic gates. Thus, the designer need not be concerned with the internal electronics of the individual gates, but only with their external logic properties. Each gate performs a specific logical operation. The outputs of gates are applied to the inputs of other gates to form a digital circuit.

In order to describe the operational properties of digital circuits, it is necessary to introduce a mathematical notation that specifies the operation of each gate and that can be used to analyze and design circuits. This binary logic system is one of a class of mathematical systems referred to generally as *Boolean algebras*. The name is in honor of the English mathematician George Boole, who in 1854 published a book introducing the mathematical theory of logic. The specific Boolean algebra we will study is used to describe the interconnection of digital gates and to design logic circuits through the manipulation of Boolean expressions. We first introduce the concept of binary logic and show its relationship to digital gates and binary signals. We then present the properties of the Boolean algebra, together with other concepts and methods useful in designing logic circuits.

Binary Logic

Binary logic deals with binary variables, which take on two discrete values, and with the operations of mathematical logic applied to these variables. The two values the variables take may be called by different names, as mentioned in Section 1-1, but for our purpose, it is convenient to think in terms of binary values and assign 1 or 0 to each variable. In the first part of this book, variables are designated by letters of the alphabet, such as A , B , C , X , Y , and Z . Later this notation will be expanded to include strings of letters, numbers, and special characters. Associated with the binary variables are three basic logical operations called AND, OR, and NOT:

1. **AND.** This operation is represented by a dot or by the absence of an operator. For example, $Z = X \cdot Y$ or $Z = XY$ is read “ Z is equal to X AND Y .” The logical operation AND is interpreted to mean that $Z = 1$ if and only if $X = 1$ and $Y = 1$; otherwise $Z = 0$. (Remember that X , Y , and Z are binary variables and can be equal to only 1 or 0.)
2. **OR.** This operation is represented by a plus symbol. For example, $Z = X + Y$ is read “ Z is equal to X OR Y ,” meaning that $Z = 1$ if $X = 1$ or if $Y = 1$, or if both $X = 1$ and $Y = 1$. $Z = 0$ if and only if $X = 0$ and $Y = 0$.
3. **NOT.** This operation is represented by a bar over the variable. For example, $Z = \bar{X}$ is read “ Z is equal to NOT X ,” meaning that Z is what X is not. In other words, if $X = 1$, then $Z = 0$; but if $X = 0$, then $Z = 1$. The NOT operation is also referred to as the *complement* operation, since it changes a 1 to 0 and a 0 to 1.

Binary logic resembles binary arithmetic, and the operations AND and OR have similarities to multiplication and addition, respectively. This is why the symbols used for AND and OR are the same as those used for multiplication and addition. However, binary logic should not be confused with binary arithmetic. One should realize that an arithmetic variable designates a number that may consist of

many digits, whereas a logic variable is always either a 1 or a 0. The following equations define the logical OR operation:

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 1$$

These resemble binary addition, except for the last operation. In binary logic, we have $1 + 1 = 1$ (read “one OR one is equal to one”), but in binary arithmetic, we have $1 + 1 = 10$ (read “one plus one is equal to two”). To avoid ambiguity, the symbol \vee is sometimes used for the OR operation instead of the $+$ symbol. But as long as arithmetic and logic operations are not mixed, each can use the $+$ symbol with its own independent meaning.

The next equations define the logical AND operation:

$$0 \cdot 0 = 0$$

$$0 \cdot 1 = 0$$

$$1 \cdot 0 = 0$$

$$1 \cdot 1 = 1$$

This operation is identical to binary multiplication, provided that we use only a single bit. Alternative symbols to the \cdot for AND and $+$ for OR, are symbols \wedge and \vee , respectively, that represent conjunctive and disjunctive operations in propositional calculus.

For each combination of the values of binary variables such as X and Y , there is a value of Z specified by the definition of the logical operation. The definitions may be listed in compact form in a truth table. A *truth table* for an operation is a table of combinations of the binary variables showing the relationship between the values that the variables take on and the values of the result of the operation. The truth tables for the operations AND, OR, and NOT are shown in Table 2-1. The tables list all possible combinations of values for two variables and the results of the operation. They clearly demonstrate the definition of the three operations.

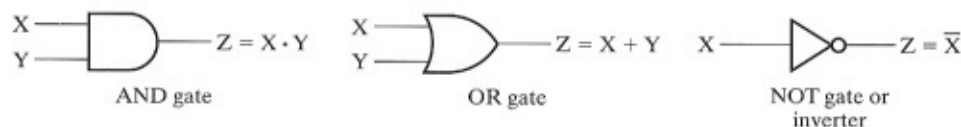
□ **TABLE 2-1**
Truth Tables for the Three Basic Logical Operations

AND			OR			NOT	
X	Y	Z = X · Y	X	Y	Z = X + Y	X	Z = \bar{X}
0	0	0	0	0	0	0	1
0	1	0	0	1	1	1	0
1	0	0	1	0	1		
1	1	1	1	1	1		

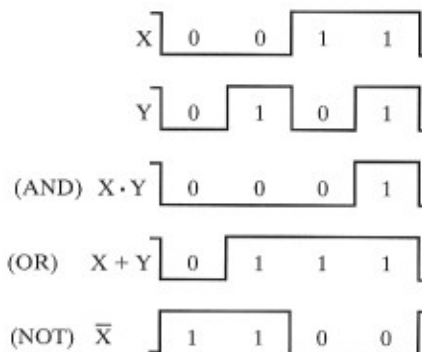
Logic Gates

Logic gates are electronic circuits that operate on one or more input signals to produce an output signal. Electrical signals such as voltages or currents exist throughout a digital system in either of two recognizable values. Voltage-operated circuits respond to two separate voltage ranges that represent a binary variable equal to logic 1 or logic 0, as illustrated in Figure 1-1. The input terminals of logic gates accept binary signals within the allowable range and respond at the output terminals with binary signals that fall within a specified range. The intermediate regions between the allowed ranges in the figure are crossed only during changes from 1 to 0 or from 0 to 1. These changes are called *transitions*, and the intermediate regions are called the *transition regions*.

The graphics symbols used to designate the three types of gates—AND, OR, and NOT—are shown in Figure 2-1(a). The gates are electronic circuits that produce the equivalents of logic-1 and logic-0 output signals in accordance with their respective truth tables if the equivalents of logic-1 and logic-0 input signals are applied. The two input signals X and Y to the AND and OR gates take on one of four possible combinations: 00, 01, 10, or 11. These input signals are shown as timing diagrams in Figure 2-1(b), together with the timing diagrams for the corresponding output signal for each type of gate. The horizontal axis of a *timing diagram* represents time, and the vertical axis shows a signal as it changes between the two possible voltage levels. The low level represents logic 0 and the high level represents logic 1. The AND gate responds with a logic-1 output signal when both input signals are logic 1. The OR gate responds with a logic-1 output signal if either



(a) Graphic symbols



(b) Timing diagram

□ **FIGURE 2-1**
Digital Logic Gates



□ **FIGURE 2-2**
Gates with More than Two Inputs

input signal is logic 1. The NOT gate is more commonly referred to as an *inverter*. The reason for this name is apparent from the response in the timing diagram. The output logic signal is an inverted version of input logic signal X .

AND and OR gates may have more than two inputs. An AND gate with three inputs and an OR gate with six inputs are shown in Figure 2-2. The three-input AND gate responds with a logic-1 output if all three inputs are logic 1. The output is logic 0 if any input is logic 0. The six-input OR gate responds with a logic 1 if any input is logic 1; its output becomes a logic 0 only when all inputs are logic 0.

2-2 BOOLEAN ALGEBRA

The Boolean algebra we present is an algebra dealing with binary variables and logic operations. The variables are designated by letters of the alphabet, and the three basic logic operations are AND, OR, and NOT (complementation). A *Boolean expression* is an algebraic expression formed by using binary variables, the constants 0 and 1, the logic operation symbols, and parentheses. A Boolean function can be described by a Boolean equation consisting of a binary variable identifying the function followed by an equal sign and a Boolean expression. Optionally, the function identifier is followed by parentheses enclosing a list of the function variables separated by commas. A *single-output Boolean function* is a mapping from each of the possible combinations of values 0 and 1 on the function variables to value 0 or 1. A *multiple-output Boolean function* is a mapping from each of the possible combinations of values 0 and 1 on the function variables to combinations of 0 and 1 on the function outputs. Consider an example Boolean equation representing function F :

$$F(X, Y, Z) = X + \bar{Y}Z$$

The two parts of the expression, X and $\bar{Y}Z$, are called *terms* of the expression for F . The function F is equal to 1 if term X is equal to 1 or if $\bar{Y}Z$ term is equal to 1 (i.e., both \bar{Y} and Z are equal to 1). Otherwise, F is equal to 0. The complement operation dictates that if $\bar{Y} = 1$, then Y must equal 0. Therefore, we can say that $F = 1$ if $X = 1$, or if $Y = 0$ and $Z = 1$. A Boolean equation expresses the logical relationship between binary variables. It is evaluated by determining the binary value of the expression for all possible combinations of values for the variables.

A Boolean function can be represented by a truth table. A *truth table* for a function is a list of all combinations of 1's and 0's that can be assigned to the binary variables and a list that shows the value of the function for each binary combination.

□ **TABLE 2-2**
Truth Table
for the Function $F = X + \bar{Y}Z$

X	Y	Z	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

The truth tables for the logic operations given in Table 2-1 are special cases of truth tables for functions. The number of rows in a truth table is 2^n , where n is the number of variables in the function. The binary combinations for the truth table are the n -bit binary numbers that correspond to counting in decimal from 0 through $2^n - 1$. Table 2-2 shows the truth table for the function $F = X + \bar{Y}Z$. There are eight possible binary combinations that assign bits to the three variables X , Y , and Z . The column labeled F contains either 0 or 1 for each of these combinations. The table shows that the function is equal to 1 if $X = 1$ and if $Y = 0$ and $Z = 1$. Otherwise, the function is equal to 0.

An algebraic expression for a Boolean function can be transformed into a circuit diagram composed of logic gates that implements the function. The logic circuit diagram for function F is shown in Figure 2-3. An inverter on input Y generates the complement, \bar{Y} . An AND gate operates on \bar{Y} and Z , and an OR gate combines X and $\bar{Y}Z$. In logic circuit diagrams, the variables of the function F are taken as the inputs of the circuit, and the binary variable F is taken as the output of the circuit. If the circuit has a single output, F is a single output function. If the circuit has multiple outputs, function F is a multiple output function with multiple equations required to represent its outputs. Circuit gates are interconnected by wires that carry logic signals. Logic circuits of this type are called *combinational logic circuits*, since the variables are “combined” by the logical operations. This is in contrast to the sequential logic to be treated in Chapter 6, in which variables are stored over time as well as being combined.



□ **FIGURE 2-3**
 Logic Circuit Diagram for $F = X + \bar{Y}Z$

There is only one way that a Boolean function can be represented in a truth table. However, when the function is in algebraic equation form, it can be expressed in a variety of ways. The particular expression used to represent the function dictates the interconnection of gates in the logic circuit diagram. By manipulating a Boolean expression according to Boolean algebraic rules, it is often possible to obtain a simpler expression for the same function. This simpler expression reduces both the number of gates in the circuit and the numbers of inputs to the gates. To see how this is done, it is necessary first to study the basic rules of Boolean algebra.

Basic Identities of Boolean Algebra

Table 2-3 lists the most basic identities of Boolean algebra. The notation is simplified by omitting the symbol for AND whenever doing so does not lead to confusion. The first nine identities show the relationship between a single variable X , its complement \bar{X} , and the binary constants 0 and 1. The next five identities, 10 through 14, have counterparts in ordinary algebra. The last three, 15 through 17, do not apply in ordinary algebra, but are useful in manipulating Boolean expressions.

The basic rules listed in the table have been arranged into two columns that demonstrate the property of duality of Boolean algebra. The *dual* of an algebraic expression is obtained by interchanging OR and AND operations and replacing 1's by 0's and 0's by 1's. An equation in one column of the table can be obtained from the corresponding equation in the other column by taking the dual of the expressions on both sides of the equals sign. For example, relation 2 is the dual of relation 1 because the OR has been replaced by an AND and the 0 by 1. It is important to note that most of the time the dual of an expression is not equal to the original expression, so that an expression usually cannot be replaced by its dual.

The nine identities involving a single variable can be easily verified by substituting each of the two possible values for X . For example, to show that $X + 0 = X$, let $X = 0$ to obtain $0 + 0 = 0$, and then let $X = 1$ to obtain $1 + 0 = 1$. Both

□ **TABLE 2-3**
Basic Identities of Boolean Algebra

1. $X + 0 = X$	2. $X \cdot 1 = X$	
3. $X + 1 = 1$	4. $X \cdot 0 = 0$	
5. $X + X = X$	6. $X \cdot X = X$	
7. $X + \bar{X} = 1$	8. $X \cdot \bar{X} = 0$	
9. $\bar{\bar{X}} = X$		
10. $X + Y = Y + X$	11. $XY = YX$	Commutative
12. $X + (Y + Z) = (X + Y) + Z$	13. $X(YZ) = (XY)Z$	Associative
14. $X(Y + Z) = XY + XZ$	15. $X + YZ = (X + Y)(X + Z)$	Distributive
16. $\overline{X + Y} = \bar{X} \cdot \bar{Y}$	17. $\overline{X \cdot Y} = \bar{X} + \bar{Y}$	DeMorgan's

equations are true according to the definition of the OR logic operation. Any expression can be substituted for the variable X in all the Boolean equations listed in the table. Thus, by identity 3 and with $X = AB + C$, we obtain

$$AB + C + 1 = 1$$

Note that identity 9 states that double complementation restores the variable to its original value. Thus, if $X = 0$, then $\overline{\overline{X}} = 1$ and $\overline{\overline{\overline{X}}} = 0 = X$.

Identities 10 and 11, the commutative laws, state that the order in which the variables are written will not affect the result when using the OR and AND operations. Identities 12 and 13, the associative laws, state that the result of applying an operation over three variables is independent of the order that is taken, and therefore, the parentheses can be removed altogether as follows:

$$X + (Y + Z) = (X + Y) + Z = X + Y + Z$$

$$X(YZ) = (XY)Z = XYZ$$

These two laws and the first distributive law, identity 14, are well known from ordinary algebra, so they should not impose any difficulty. The second distributive law, given by identity 15, is the dual of the ordinary distributive law and does not hold in ordinary algebra. As illustrated previously, each variable in an identity can be replaced by a Boolean expression, and the identity still holds. Thus, consider the expression $(A + B)(A + CD)$. Letting $X = A$, $Y = B$, and $Z = CD$, and applying the second distributive law, we obtain

$$(A + B)(A + CD) = A + BCD$$

The last two identities in Table 2-3,

$$\overline{X + Y} = \overline{X} \cdot \overline{Y} \text{ and } \overline{X \cdot Y} = \overline{X} + \overline{Y}$$

are referred to as DeMorgan's theorem. This is a very important theorem and is used to obtain the complement of an expression and of the corresponding function. DeMorgan's theorem can be illustrated by means of truth tables that assign all the possible binary values to X and Y . Table 2-4 shows two truth tables that verify the first part of DeMorgan's theorem. In A, we evaluate $\overline{X + Y}$ for all possible values of X and Y . This is done by first evaluating $X + Y$ and then taking its complement. In B, we evaluate \overline{X} and \overline{Y} and then AND them together. The result is the same

□ **TABLE 2-4**
Truth Tables to Verify DeMorgan's Theorem

A)	X	Y	X + Y	$\overline{X + Y}$	B)	X	Y	\overline{X}	\overline{Y}	$\overline{X} \cdot \overline{Y}$
	0	0	0	1		0	0	1	1	1
	0	1	1	0		0	1	1	0	0
	1	0	1	0		1	0	0	1	0
	1	1	1	0		1	1	0	0	0

for the four binary combinations of X and Y , which verifies the identity of the equation.

Note the order in which the operations are performed when evaluating an expression. In part B of the table, the complement over a single variable is evaluated first, followed by the AND operation, just as in ordinary algebra with multiplication and addition. In part A, the OR operation is evaluated first. Then, noting that the complement over an expression such as $X + Y$ is considered as specifying NOT ($X + Y$), we evaluate the expression within the parentheses and take the complement of the result. It is customary to exclude the parentheses when complementing an expression, since a bar over the entire expression joins it together. Thus, $\overline{(X + Y)}$ is expressed as $\overline{X + Y}$ when designating the complement of $X + Y$.

DeMorgan's theorem can be extended to three or more variables. The general DeMorgan's theorem can be expressed as

$$\begin{aligned}\overline{X_1 + X_2 + \dots + X_n} &= \overline{X_1} \overline{X_2} \dots \overline{X_n} \\ \overline{X_1 X_2 \dots X_n} &= \overline{X_1} + \overline{X_2} + \dots + \overline{X_n}\end{aligned}$$

Observe that the logic operation changes from OR to AND or from AND to OR. In addition, the complement is removed from the entire expression and placed instead over each variable. For example,

$$\overline{A + B + C + D} = \overline{A} \overline{B} \overline{C} \overline{D}$$

Algebraic Manipulation

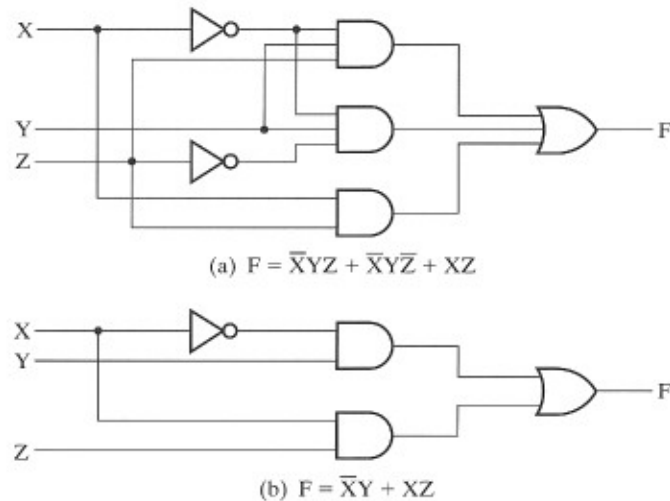
Boolean algebra is a useful tool for simplifying digital circuits. Consider, for example, the Boolean function represented by

$$F = \overline{X}YZ + \overline{X}Y\overline{Z} + XZ$$

The implementation of this equation with logic gates is shown in Figure 2-4(a). Input variables X and Z are complemented with inverters to obtain \overline{X} and \overline{Z} . The three terms in the expression are implemented with three AND gates. The OR gate forms the logical OR of the terms. Now consider a simplification of the expression for F by applying some of the identities listed in Table 2-3:

$$\begin{aligned}F &= \overline{X}YZ + \overline{X}Y\overline{Z} + XZ \\ &= \overline{X}Y(Z + \overline{Z}) + XZ && \text{by identity 14} \\ &= \overline{X}Y \cdot 1 + XZ && \text{by identity 7} \\ &= \overline{X}Y + XZ && \text{by identity 2}\end{aligned}$$

The expression is reduced to only two terms and can be implemented with gates as shown in Figure 2-4(b). It is obvious that the circuit in (b) is simpler than the one in (a), yet, both implement the same function. It is possible to use a truth table to verify that the two implementations are equivalent. This is shown in Table 2-5. As expressed in Figure 2-4(a), the function is equal to 1 if $X = 0$, $Y = 1$, and $Z = 1$; if $X = 0$, $Y = 1$, and $Z = 0$; or if X and Z are both 1. This produces the



□ **FIGURE 2-4**
Implementation of Boolean Function with Gates

□ **TABLE 2-5**
Truth Table for Boolean Function

X	Y	Z	(a) F	(b) F
0	0	0	0	0
0	0	1	0	0
0	1	0	1	1
0	1	1	1	1
1	0	0	0	0
1	0	1	1	1
1	1	0	0	0
1	1	1	1	1

four 1's for F in part (a) of the table. As expressed in Figure 2-4(b), the function is equal to 1 if $X = 0$ and $Y = 1$ or if $X = 1$ and $Z = 1$. This produces the same four 1's in part (b) of the table. Since both expressions produce the same truth table, they are equivalent. Therefore, the two circuits have the same output for all possible binary combinations of the three input variables. Each circuit implements the same function, but the one with fewer gates is preferable because it requires fewer components.

When a Boolean equation is implemented with logic gates, each term requires a gate, and each variable within the term designates an input to the gate. We define a *literal* as a single variable within a term that may or may not be complemented. The expression for the function in Figure 2-4(a) has three terms and eight literals; the one in Figure 2-4(b) has two terms and four literals. By

reducing the number of terms, the number of literals, or both in a Boolean expression, it is often possible to obtain a simpler circuit. Boolean algebra is applied to reduce an expression for the purpose of obtaining a simpler circuit. For highly complex functions, finding the best expression based on counts of terms and literals is very difficult, even by the use of computer programs. Certain methods, however, for reducing expressions are often included in computer tools for synthesizing logic circuits. These methods can obtain good, if not the best, solutions. The only manual method for the general case is a cut-and-try procedure employing the basic relations and other manipulations that become familiar with use. The following examples use identities from Table 2-3 to illustrate a few of the possibilities:

1. $X + XY = X(1 + Y) = X$
2. $XY + X\bar{Y} = X(Y + \bar{Y}) = X$
3. $X + \bar{X}Y = (X + \bar{X})(X + Y) = X + Y$

Note that the intermediate step $X = X \cdot 1$ has been omitted when X is factored out in equation 1. The relationship $1 + Y = 1$ is useful for eliminating redundant terms, as is done with the term XY in this same equation. The relation $Y + \bar{Y} = 1$ is useful for combining two terms, as is done in equation 2. The two terms being combined must be identical except for one variable, and that variable must be complemented in one term and not complemented in the other. Equation 3 is simplified by means of the second distributive law (identity 15 in Table 2-3). The following are three more examples of simplifying Boolean expressions:

4. $X(X + Y) = X + XY = X$
5. $(X + Y)(X + \bar{Y}) = X + Y\bar{Y} = X$
6. $X(\bar{X} + Y) = X\bar{X} + XY = XY$

Note that the intermediate steps $XX = X = X \cdot 1$ have been omitted during the manipulation of equation 4. The expression in equation 5 is simplified by means of the second distributive law. Here again, we omit the intermediate steps $Y\bar{Y} = 0$ and $X + 0 = X$.

Equations 4 through 6 are the duals of equations 1 through 3. Remember that the dual of an expression is obtained by changing AND to OR and OR to AND throughout (and 1's to 0's and 0's to 1's if they appear in the expression). The *duality principle* of Boolean algebra states that a Boolean equation remains valid if we take the dual of the expressions on both sides of the equals sign. Therefore, equations 4, 5, and 6 can be obtained by taking the dual of equations 1, 2, and 3, respectively.

Along with the results just given in equations 1 through 6, the following *consensus theorem* is useful when simplifying Boolean expressions:

$$XY + \bar{X}Z + YZ = XY + \bar{X}Z$$

The theorem shows that the third term, YZ , is redundant and can be eliminated. Note that Y and Z are associated with X and \bar{X} in the first two terms and appear

together in the term that is eliminated. The proof of the consensus theorem is obtained by first ANDing YZ with $(X + \bar{X}) = 1$ and proceeds as follows:

$$\begin{aligned}
 XY + \bar{X}Z + YZ &= XY + \bar{X}Z + YZ(X + \bar{X}) \\
 &= XY + \bar{X}Z + XYZ + \bar{X}YZ \\
 &= XY + XYZ + \bar{X}Z + \bar{X}YZ \\
 &= XY(1 + Z) + \bar{X}Z(1 + Y) \\
 &= XY + \bar{X}Z
 \end{aligned}$$

The dual of the consensus theorem is

$$(X + Y)(\bar{X} + Z)(Y + Z) = (X + Y)(\bar{X} + Z)$$

The following example shows how the consensus theorem can be applied in manipulating a Boolean expression:

$$\begin{aligned}
 (A + B)(\bar{A} + C) &= A\bar{A} + AC + \bar{A}B + BC \\
 &= AC + \bar{A}B + BC \\
 &= AC + \bar{A}B
 \end{aligned}$$

Note that $A\bar{A} = 0$ and $0 + AC = AC$. The redundant term eliminated in the last step by the consensus theorem is BC .

Complement of a Function

The complement representation for a function F , \bar{F} , is obtained from an interchange of 1's to 0's and 0's to 1's for the values of F in the truth table. The complement of a function can be derived algebraically by applying DeMorgan's theorem. The generalized form of this theorem states that the complement of an expression is obtained by interchanging AND and OR operations and complementing each variable and constant, as shown in Example 2-1.

EXAMPLE 2-1 Complementing Functions

Find the complement of each of the functions represented by the equations $F_1 = \bar{X}YZ + \bar{X}\bar{Y}Z$ and $F_2 = X(\bar{Y}\bar{Z} + YZ)$. Applying DeMorgan's theorem as many times as necessary, we obtain the complements as follows:

$$\begin{aligned}
 \bar{F}_1 &= \overline{\bar{X}YZ + \bar{X}\bar{Y}Z} = \overline{(\bar{X}YZ)} \cdot \overline{(\bar{X}\bar{Y}Z)} \\
 &= (X + \bar{Y} + Z)(X + Y + \bar{Z}) \\
 \bar{F}_2 &= \overline{X(\bar{Y}\bar{Z} + YZ)} = \bar{X} + \overline{(\bar{Y}\bar{Z} + YZ)}
 \end{aligned}$$

$$\begin{aligned}
 &= \bar{X} + (\bar{Y}\bar{Z} \cdot \bar{Y}\bar{Z}) \\
 &= \bar{X} + (Y + Z)(\bar{Y} + \bar{Z})
 \end{aligned}$$

A simpler method for deriving the complement of a function is to take the dual of the function equation and complement each literal. This method follows from the generalization of DeMorgan's theorem. Remember that the dual of an expression is obtained by interchanging AND and OR operations and 1's and 0's. To avoid confusion in handling complex functions, adding parentheses around terms before taking the dual is helpful, as illustrated in the next example.

EXAMPLE 2-2 Complementing Functions by Using Duals

Find the complements of the functions in Example 2-1 by taking the duals of their equations and complementing each literal.

We begin with

$$F_1 = \bar{X}Y\bar{Z} + \bar{X}\bar{Y}Z = (\bar{X}Y\bar{Z}) + (\bar{X}\bar{Y}Z)$$

The dual of F_1 is

$$(\bar{X} + Y + \bar{Z})(\bar{X} + \bar{Y} + Z)$$

Complementing each literal, we have

$$(X + \bar{Y} + Z)(X + Y + \bar{Z}) = \bar{F}_1$$

Now,

$$F_2 = X(\bar{Y}\bar{Z} + YZ) = X((\bar{Y}\bar{Z}) + (YZ))$$

The dual of F_2 is

$$X + (\bar{Y} + \bar{Z})(Y + Z)$$

Complementing each literal yields

$$\bar{X} + (Y + Z)(\bar{Y} + \bar{Z}) = \bar{F}_2.$$

2-3 STANDARD FORMS

A Boolean function expressed algebraically can be written in a variety of ways. There are, however, specific ways of writing algebraic equations that are considered to be standard forms. The standard forms facilitate the simplification procedures for Boolean expressions and frequently result in more desirable logic circuits.

The standard forms contain *product terms* and *sum terms*. An example of a product term is $X\bar{Y}Z$. This is a logical product consisting of an AND operation among three literals. An example of a sum term is $X + Y + \bar{Z}$. This is a logical sum consisting of an OR operation among the literals. It must be realized that the

words “product” and “sum” do not imply arithmetic operations in Boolean algebra; instead, they specify the logical operations AND and OR, respectively.

Minterms and Maxterms

It has been shown that a truth table defines a Boolean function. An algebraic expression representing the function is derived from the table by finding the logical sum of all product terms for which the function assumes the binary value 1. A product term in which all the variables appear exactly once, either complemented or uncomplemented, is called a *minterm*. Its characteristic property is that it represents exactly one combination of the binary variables in a truth table. It has the value 1 for that combination and 0 for all others. There are 2^n distinct minterms for n variables. The four minterms for the two variables X and Y are $\bar{X}\bar{Y}$, $\bar{X}Y$, $X\bar{Y}$, and XY . The eight minterms for the three variables X , Y , and Z are listed in Table 2-6. The binary numbers from 000 to 111 are listed under the variables. For each binary combination, there is a related minterm. Each minterm is a product term of exactly three literals. A literal is a complemented variable if the corresponding bit of the related binary combination is 0 and is an uncomplemented variable if it is 1. A symbol m_j for each minterm is also shown in the table, where the subscript j denotes the decimal equivalent of the binary combination for which the minterm has the value 1. This list of minterms for any given n variables can be formed in a similar manner from a list of the binary numbers from 0 through $2^n - 1$. In addition, the truth table for each minterm is given in the right half of the table. These truth tables clearly show that each minterm is 1 for the corresponding binary combination and 0 for all other combinations. Such truth tables will be helpful later in using minterms to form Boolean expressions.

A sum term that contains all the variables in complemented or uncomplemented form is called a *maxterm*. Again, it is possible to formulate 2^n maxterms with n variables. The eight maxterms for three variables are listed in Table 2-7. Each maxterm is a logical sum of the three variables, with each variable being complemented if the corresponding bit of the binary number is 1 and uncomplemented

□ TABLE 2-6
Minterms for Three Variables

X	Y	Z	Product Term	Symbol	m_0	m_1	m_2	m_3	m_4	m_5	m_6	m_7
0	0	0	$\bar{X}\bar{Y}\bar{Z}$	m_0	1	0	0	0	0	0	0	0
0	0	1	$\bar{X}\bar{Y}Z$	m_1	0	1	0	0	0	0	0	0
0	1	0	$\bar{X}Y\bar{Z}$	m_2	0	0	1	0	0	0	0	0
0	1	1	$\bar{X}YZ$	m_3	0	0	0	1	0	0	0	0
1	0	0	$X\bar{Y}\bar{Z}$	m_4	0	0	0	0	1	0	0	0
1	0	1	$X\bar{Y}Z$	m_5	0	0	0	0	0	1	0	0
1	1	0	$XY\bar{Z}$	m_6	0	0	0	0	0	0	1	0
1	1	1	XYZ	m_7	0	0	0	0	0	0	0	1

□ **TABLE 2-7**
Maxterms for Three Variables

X	Y	Z	Sum Term	Symbol	M ₀	M ₁	M ₂	M ₃	M ₄	M ₅	M ₆	M ₇
0	0	0	$X + Y + Z$	M ₀	0	1	1	1	1	1	1	1
0	0	1	$X + Y + \bar{Z}$	M ₁	1	0	1	1	1	1	1	1
0	1	0	$X + \bar{Y} + Z$	M ₂	1	1	0	1	1	1	1	1
0	1	1	$X + \bar{Y} + \bar{Z}$	M ₃	1	1	1	0	1	1	1	1
1	0	0	$\bar{X} + Y + Z$	M ₄	1	1	1	1	0	1	1	1
1	0	1	$\bar{X} + Y + \bar{Z}$	M ₅	1	1	1	1	1	0	1	1
1	1	0	$\bar{X} + \bar{Y} + Z$	M ₆	1	1	1	1	1	1	0	1
1	1	1	$\bar{X} + \bar{Y} + \bar{Z}$	M ₇	1	1	1	1	1	1	1	0

if it is 0. The symbol for a maxterm is M_j , where j denotes the decimal equivalent of the binary combination for which the maxterm has the value 0. In the right half of the table, the truth table for each maxterm is given. Note that the value of the maxterm is 0 for the corresponding combination and 1 for all other combinations. It is now clear where the terms “minterm” and “maxterm” come from: a minterm is a function, not equal to 0, having the minimum number of 1's in its truth table; a maxterm is a function, not equal to 1, having the maximum of 1's in its truth table. Note from Table 2-6 and Table 2-7 that a minterm and maxterm with the same subscript are the complements of each other; that is, $M_j = \bar{m}_j$. For example, for $j = 3$, we have

$$\bar{m}_3 = \overline{\bar{X}YZ} = X + \bar{Y} + \bar{Z} = M_3$$

A Boolean function can be represented algebraically from a given truth table by forming the logical sum of all the minterms that produce a 1 in the function. This expression is called a *sum of minterms*. Consider the Boolean function F in Table 2-8(a). The function is equal to 1 for each of the following binary combinations of the variables X , Y , and Z : 000, 010, 101 and 111. These combinations correspond to minterms 0, 2, 5, and 7. By examining Table 2-8 and the truth tables for these minterms in Table 2-6, it is evident that the function F can be expressed algebraically as the logical sum of the stated minterms:

$$F = \bar{X}\bar{Y}\bar{Z} + \bar{X}Y\bar{Z} + X\bar{Y}Z + XYZ = m_0 + m_2 + m_5 + m_7$$

This can be further abbreviated by listing only the decimal subscripts of the minterms:

$$F(X, Y, Z) = \Sigma m(0, 2, 5, 7)$$

The symbol Σ stands for the logical sum (Boolean OR) of the minterms. The numbers following it represent the minterms of the function. The letters in parentheses following F form a list of the variables in the order taken when the minterms are converted to product terms.

□ TABLE 2-8
Boolean Functions of Three Variables

(a)	X	Y	Z	F	\bar{F}	(b)	X	Y	Z	E
	0	0	0	1	0		0	0	0	1
	0	0	1	0	1		0	0	1	1
	0	1	0	1	0		0	1	0	1
	0	1	1	0	1		0	1	1	0
	1	0	0	0	1		1	0	0	1
	1	0	1	1	0		1	0	1	1
	1	1	0	0	1		1	1	0	0
	1	1	1	1	0		1	1	1	0

Now consider the complement of a Boolean function. The binary values of \bar{F} in Table 2-8(a) are obtained by changing 1's to 0's and 0's to 1's in the values of F . Taking the logical sum of minterms of \bar{F} , we obtain

$$\bar{F} = \bar{X}\bar{Y}Z + \bar{X}YZ + X\bar{Y}\bar{Z} + XY\bar{Z} = m_1 + m_3 + m_4 + m_6$$

or, in abbreviated form,

$$\bar{F}(X, Y, Z) = \Sigma m(1, 3, 4, 6)$$

Note that the minterm numbers for \bar{F} are the ones missing from the list of the minterm numbers of F . We now take the complement of \bar{F} to obtain F :

$$\begin{aligned} F &= \overline{m_1 + m_3 + m_4 + m_6} = \overline{m_1} \cdot \overline{m_3} \cdot \overline{m_4} \cdot \overline{m_6} \\ &= M_1 \cdot M_3 \cdot M_4 \cdot M_6 \quad (\text{Since } \overline{m_j} = M_j) \\ &= (X + Y + \bar{Z})(X + \bar{Y} + \bar{Z})(\bar{X} + Y + Z)(\bar{X} + \bar{Y} + Z) \end{aligned}$$

This shows the procedure for expressing a Boolean function as a *product of maxterms*. The abbreviated form for this product is

$$F(X, Y, Z) = \Pi M(1, 3, 4, 6)$$

where symbol Π denotes the logical product (Boolean AND) of the maxterms whose numbers are listed in parentheses. Note that the decimal numbers included in the product of maxterms will always be the same as the minterm list of the complemented function, such as (1, 3, 4, 6) in the foregoing example. Maxterms are seldom used directly when dealing with Boolean functions, as we can always replace them with the minterm list of \bar{F} .

The following is a summary of the most important properties of minterms:

1. There are 2^n minterms for n Boolean variables. These minterms can be evaluated from the binary numbers from 0 to $2^n - 1$.
2. Any Boolean function can be expressed as a logical sum of minterms.

3. The complement of a function contains those minterms not included in the original function.
4. A function that includes all the 2^n minterms is equal to logic 1.

A function that is not in the sum-of-minterms form can be converted to that form by means of a truth table, since the truth table always specifies the minterms of the function. Consider, for example, the Boolean function

$$E = \bar{Y} + \bar{X}\bar{Z}$$

The expression is not in sum-of-minterms form, because each term does not contain all three variables X , Y , and Z . The truth table for this function is listed in Table 2-8(b). From the table, we obtain the minterms of the function:

$$E(X, Y, Z) = \Sigma m(0, 1, 2, 4, 5)$$

The minterms for the complement of E are given by

$$\bar{E}(X, Y, Z) = \Sigma m(3, 6, 7)$$

Note that the total number of minterms in E and \bar{E} is equal to eight, since the function has three variables, and three variables produce a total of eight minterms. With four variables, there will be a total of 16 minterms, and for two variables, there will be 4 minterms. An example of a function that includes all the minterms is

$$G(X, Y) = \Sigma m(0, 1, 2, 3) = 1$$

Since G is a function of two variables and contains all four minterms, it is always equal to logic 1.

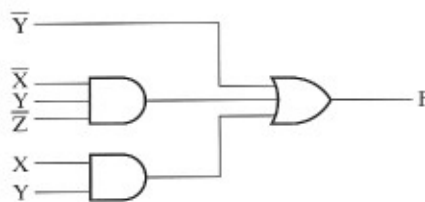
Sum of Products

The sum-of-minterms form is a standard algebraic expression that is obtained directly from a truth table. The expression so obtained contains the maximum number of literals in each term and usually has more product terms than necessary. This is because, by definition, each minterm must include all the variables of the function, complemented or uncomplemented. Once the sum of minterms is obtained from the truth table, the next step is to try to simplify the expression to see whether it is possible to reduce the number of product terms and the number of literals in the terms. The result is a simplified expression in *sum-of-products* form. This is an alternative standard form of expression that contains product terms with one, two, or any number of literals. An example of a Boolean function expressed as a sum of products is

$$F = \bar{Y} + \bar{X}Y\bar{Z} + XY$$

The expression has three product terms, the first with one literal, the second with three literals, and the third with two literals.

The logic diagram for a sum-of-products form consists of a group of AND gates followed by a single OR gate, as shown in Figure 2-5. Each product term



□ **FIGURE 2-5**
Sum-of-Products Implementation

requires an AND gate, except for a term with a single literal. The logical sum is formed with an OR gate that has single literals and the outputs of the AND gates as inputs. It is assumed that the input variables are directly available in their complemented and uncomplemented forms, so inverters are not included in the diagram. The AND gates followed by the OR gate form a circuit configuration referred to as a *two-level implementation* or *two-level circuit*.

If an expression is not in sum-of-products-form, it can be converted to the standard form by means of the distributive laws. Consider the expression

$$F = AB + C(D + E)$$

This is not in sum-of-products form, because the term $D + E$ is part of a product, but is not a single literal. The expression can be converted to a sum of products by applying the appropriate distributive law as follows:

$$F = AB + C(D + E) = AB + CD + CE$$

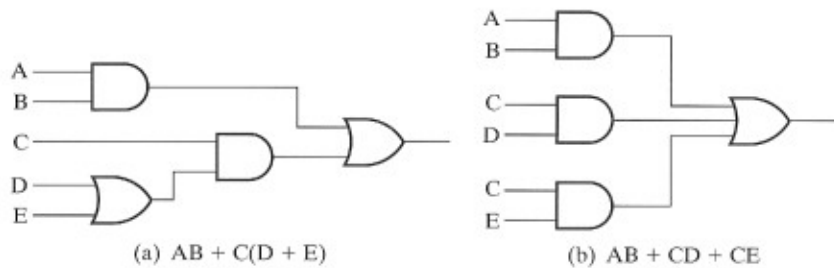
The function F is implemented in a nonstandard form in Figure 2-6(a). This requires two AND gates and two OR gates. There are three levels of gating in the circuit. F is implemented in sum-of-products form in Figure 2-6(b). This circuit requires three AND gates and an OR gate and uses two levels of gating. The decision as to whether to use a two-level or multiple-level (three levels or more) implementation is complex. Among the issues involved are the number of gates, number of gate inputs, and the amount of delay between the time the input values are set and the time the resulting output values appear. Two-level implementations are the natural form for certain implementation technologies, as we will see in Chapter 4.

Product of Sums

Another standard form of expressing Boolean functions algebraically is the *product of sums*. This form is obtained by forming a logical product of sum terms. Each logical sum term may have any number of distinct literals. An example of a function expressed in product-of-sums form is

$$F = X(\bar{Y} + Z)(X + Y + \bar{Z})$$

This expression has sum terms of one, two, and three literals. The sum terms perform an OR operation, and the product is an AND operation.

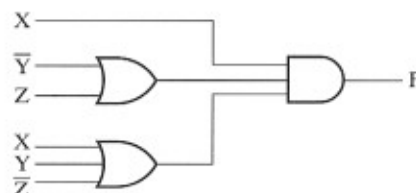


□ **FIGURE 2-6**
Three-Level and Two-Level Implementation

The gate structure of the product-of-sums expression consists of a group of OR gates for the sum terms (except for a single literal term), followed by an AND gate. This is shown in Figure 2-7 for the preceding function F . As with the sum of products, this standard type of expression results in a two-level gating structure.

2-4 TWO-LEVEL CIRCUIT OPTIMIZATION

The complexity of the digital logic gates that implement a Boolean function is directly related to the algebraic expression from which the function is implemented. Although the truth table representation of a function is unique, when expressed algebraically, the function appears in many different forms. Boolean expressions may be simplified by algebraic manipulation as discussed in Section 2-2. However, this procedure of simplification is awkward because it lacks specific rules to predict each succeeding step in the manipulative process and it is difficult to determine whether the simplest expression has been achieved. By contrast, the map method provides a straightforward procedure for optimizing Boolean functions of up to four variables. Maps for five and six variables can be drawn as well, but are more cumbersome to use. The map is also known as the *Karnaugh map*, or *K-map*. The map is a diagram made up of squares, with each square representing one minterm of the function. Since any Boolean function can be expressed as a sum of minterms, it follows that a Boolean function is recognized graphically in the map by those squares whose minterms are included in the function. In fact, the map presents a visual diagram of all possible ways a function may be expressed in a standard form. By recognizing various patterns, the user can derive alternative algebraic



□ **FIGURE 2-7**
Product-of-Sums Implementation

expressions for the same function, from which the simplest can be selected. The optimized expressions produced by the map are always in sum-of-products or product-of-sums form. Thus, maps handle optimization for two-level implementations, but do not apply directly to possible simpler implementations for the general case with three or more levels. Initially, this section covers sum-of-products optimization and, later, applies it to performing product-of-sums optimization.

Cost Criteria

In the prior section, counting literals and terms was mentioned as a way of measuring the simplicity of a logic circuit. We introduce two cost criteria to formalize this concept.

The first criterion is *literal cost*, the number of literal appearances in a Boolean expression corresponding exactly to the logic diagram. For example, for the circuits in Figure 2-6, the corresponding Boolean expressions are

$$F = AB + C(D + E) \text{ and } F = AB + CD + CE$$

There are five literal appearances in the first equations and six literal appearances in the second equation, so the first equation is the simplest in terms of literal cost. Literal cost has the advantage that it is very simple to evaluate by counting literal appearances. It does not, however, represent circuit complexity accurately in all cases, even for the comparison of different implementations of the same logic function. The following Boolean equations, both for function G , illustrate this situation:

$$G = ABCD + \overline{ABCD} \text{ and } G = (\overline{A} + B)(\overline{B} + C)(\overline{C} + D)(\overline{D} + A)$$

The implementations represented by these equations both have a literal cost of eight. But, the first equation has two terms and the second equation has four terms. This suggests that the first equation has a lower cost than the second equation.

To capture the difference illustrated, we define *gate input cost* as the number of inputs to the gates in the implementation corresponding exactly to the given equation or equations. This cost can be determined easily from the logic diagram by simply counting the total number of inputs to the gates in the logic diagram. For sum-of-products or product-of-sums equations, it can be found from the equation by finding the sum of

- (1) all literal appearances,
- (2) the number of terms excluding terms that consist only of a single literal, and, optionally,
- (3) the number of distinct complemented single literals.

In (1), all gate inputs from outside the circuit are represented. In (2), all gate inputs within the circuit, except for those to inverters are represented and in (3), inverters needed to complement the input variables are counted in the event that complemented input variables are not provided. For the two preceding equations, excluding the count from (3), the respective gate input counts are $8 + 2 = 10$ and $8 + 4 = 12$.

Including the count from (3), that of input inverters, the respective counts are 14 and 16. So the first equation for G has a lower gate input cost even though the literal costs are equal.

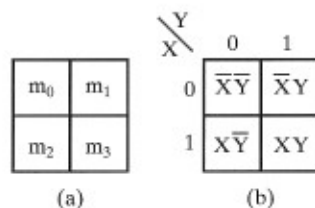
Gate input cost is currently a good measure for contemporary logic implementations since it is proportional to the number of transistors and wires used in implementing a logic circuit. Representation of gate inputs becomes particularly important in measuring cost for circuits with more than two levels. Typically, as the number of levels increases, literal cost represents a smaller proportion of the actual circuit cost since more and more gates have no inputs from outside the circuit itself. Later, in Figure 2-29, we introduce complex gate types for which evaluation of the gate input cost from an equation is invalid, since the correspondence between the AND, OR and NOT operations in the equation and the gates in the circuit can no longer be established. In such cases, as well as for equation forms more complex than sum-of-products and product-of-sums, the gate input count must be determined directly from the implementation.

Regardless of the cost criteria used, we see later that the simplest expression is not necessarily unique. It is sometimes possible to find two or more expressions that satisfy the cost criterion applied. In that case, either solution is satisfactory from the cost standpoint.

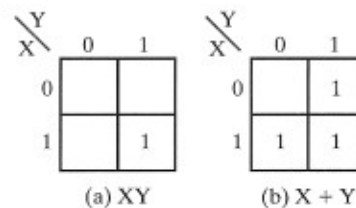
Two-Variable Map

There are four minterms for a Boolean function with two variables. Hence, the two-variable map consists of four squares, one for each minterm, as shown in Figure 2-8(a). The map is redrawn in Figure 2-8(b) to show the relationship between the squares and the two variables X and Y . The 0 and 1 marked on the left side and the top of the map designate the values of the variables. The variable X appears complemented in row 0 and uncomplemented in row 1. Similarly, Y appears complemented in column 0 and uncomplemented in column 1. Note that the four combinations of these binary values correspond to the truth table rows associated with the four minterms.

A function of two variables can be represented in a map by marking the squares that correspond to the minterms of the function. As an example, the function XY is shown in Figure 2-9(a). Since XY is equal to minterm m_3 , a 1 is placed



□ **FIGURE 2-8**
Two-Variable Map



□ **FIGURE 2-9**
Representation of Functions in the Map

inside the square that belongs to m_3 . Figure 2-9(b) shows the map for the logical sum of three minterms:

$$m_1 + m_2 + m_3 = \bar{X}Y + X\bar{Y} + XY = X + Y$$

The optimized expression $X + Y$ is determined from the two-square area for the variable X in the second row and the two-square area for Y in the second column. Together, these two areas enclose the three squares belonging to X or Y . This simplification can be justified by algebraic manipulation:

$$\bar{X}Y + X\bar{Y} + XY = \bar{X}Y + X(\bar{Y} + Y) = (\bar{X} + X)(Y + X) = X + Y$$

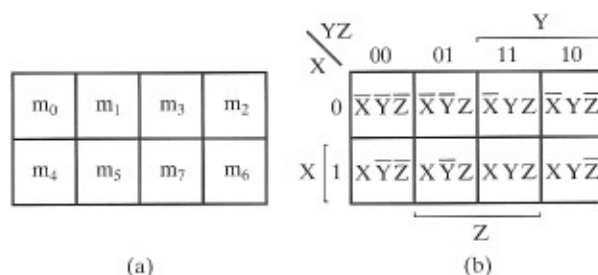
The exact procedure for combining squares in the map will be clarified in the examples that follow.

Three-Variable Map

There are eight minterms for three binary variables. Therefore, a three-variable map consists of eight squares, as shown in Figure 2-10. The map drawn in part (b) is marked with binary numbers for each row and each column to show the binary values of the minterms. Note that the numbers along the columns do not follow the binary count sequence. The characteristic of the listed sequence is that only one bit changes in value from one adjacent column to the next, which corresponds to the Gray code introduced in Chapter 1.

A minterm square can be located in the map in two ways. First, we can memorize the numbers listed in Figure 2-10(a) for each minterm location, or we can refer to the binary numbers along the rows and columns in Figure 2-10(b). For example, the square assigned to m_5 corresponds to row 1 and column 01. When these two numbers are combined, they give the binary number 101, whose decimal equivalent is 5.

Another way of looking at square $m_5 = X\bar{Y}Z$ is to consider it to be in the row marked X and the column belonging to $\bar{Y}Z$ (column 01). Note that there are four squares where each variable is equal to 1 and four where each is equal to 0.



□ **FIGURE 2-10**
Three-Variable Map

The variable appears uncomplemented in the four squares where it is equal to 1 and complemented in the four squares where it is equal to 0. For convenience, we write the variable name along the four squares where it is uncomplemented. After one becomes familiar with maps, the use of the variable names alone is sufficient to label the map regions. To this end, it is important to note the location of these labels to obtain all minterms on the map.

In the two-variable map, the function XY demonstrated that a function or a term for a function can consist of a single square of the map. But to achieve simplification, we need to consider multiple squares corresponding to product terms. To understand how combining squares simplifies Boolean functions, we must recognize the basic property possessed by adjacent squares: Any two adjacent squares placed horizontally or vertically (but not diagonally) to form a rectangle correspond to minterms that differ in only a single variable. The single variable appears uncomplemented in one square and complemented in the other. For example, m_5 and m_7 lie in two adjacent squares. Variable Y is complemented in m_5 and uncomplemented in m_7 , while the other two variables match in both squares. The logical sum of two such adjacent minterms can be simplified into a single product term of two variables:

$$m_5 + m_7 = X\bar{Y}Z + XYZ = XZ(\bar{Y} + Y) = XZ$$

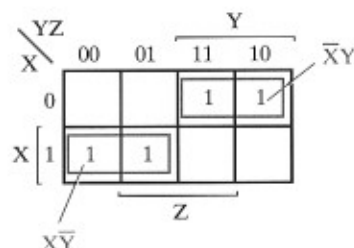
Here the two squares differ in the variable Y , which can be removed when the logical sum (OR) of the two minterms is formed. Thus, on a 3-variable map, any two minterms in adjacent squares that are ORed together produce a product term of two variables. This is shown in Example 2-3.

EXAMPLE 2-3 Simplifying a Boolean Function Using a Map

Simplify the Boolean function

$$F(X, Y, Z) = \Sigma m(2, 3, 4, 5)$$

First, a 1 is marked in each minterm that represents the function. This is shown in Figure 2-11, where the squares for minterms 010, 011, 100, and 101 are marked with 1's. For convenience, all of the remaining squares for which the function has value 0 are left blank rather than entering the 0's. The next step is to explore collections of squares on the map representing product terms to be considered for the simplified expression. We call such objects *rectangles*, since their shape is that of a rectangle (including, of course, a square). Rectangles that correspond to product terms are restricted, however, to contain numbers of squares that are powers of 2, such as 1, 2, 4, 8, So our goal is to find the fewest such rectangles that include all of the minterms marked with 1's. This will give the fewest product terms. In the map in the figure, two rectangles enclose all four squares containing 1's. The upper right rectangle represents the product term $\bar{X}Y$. This is determined by observing that the rectangle is in row 0, corresponding to \bar{X} , and the last two columns, corresponding to Y . Similarly, the lower left rectangle represents the product term $X\bar{Y}$. (The second row represents X and the two left columns represent \bar{Y} .) Since these

□ **FIGURE 2-11**

Map for Example 2-3: $F(X, Y, Z) = \Sigma m(2, 3, 4, 5) = \bar{X}Y + X\bar{Y}$

two rectangles include all of the 1's in the map, the logical sum of the corresponding two product terms gives the optimized expression for F :

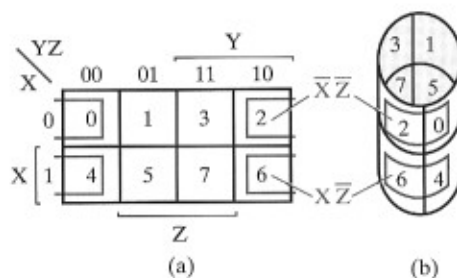
$$F = \bar{X}Y + X\bar{Y}$$

In some cases, two squares in the map are adjacent and form a rectangle of size two, even though they do not touch each other. For example, in Figure 2-10, m_0 is adjacent to m_2 and m_4 is adjacent to m_6 because the minterms differ by one variable. This can be readily verified algebraically:

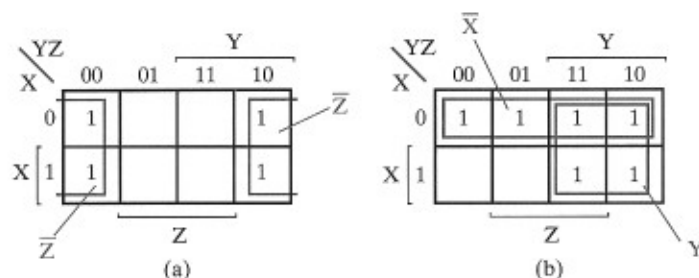
$$m_0 + m_2 = \bar{X}\bar{Y}\bar{Z} + \bar{X}Y\bar{Z} = \bar{X}\bar{Z}(\bar{Y} + Y) = \bar{X}\bar{Z}$$

$$m_4 + m_6 = X\bar{Y}\bar{Z} + XY\bar{Z} = X\bar{Z}(\bar{Y} + Y) = X\bar{Z}$$

The rectangles corresponding to these two product terms, $\bar{X}\bar{Z}$ and $X\bar{Z}$, are shown on the map in Figure 2-12(a). Based on the location of these rectangles, we must modify the definition of adjacent squares to include this and other, similar cases. We do so by considering the map as being drawn on a cylinder, as shown in Figure 2-12(b), where the right and left edges touch each other to correctly establish minterm adjacencies and form the rectangles. In the maps in Figure 2-12, we have simply used numbers rather than m 's to represent the minterms. Both of these notations will be used freely.

□ **FIGURE 2-12**

Three-Variable Map: Flat and on a Cylinder to Show Adjacent Squares



□ **FIGURE 2-13**
Product Terms Using Four Minterms

A four-square rectangle represents a product term that is the logical sum of four minterms. For the three-variable case, such a product term is only one literal. As an example, the logical sum of the four adjacent minterms 0, 2, 4, and 6 reduces to a single literal term \bar{Z} :

$$\begin{aligned}
 m_0 + m_2 + m_4 + m_6 &= \bar{X}\bar{Y}\bar{Z} + \bar{X}Y\bar{Z} + X\bar{Y}\bar{Z} + XY\bar{Z} \\
 &= \bar{X}\bar{Z}(\bar{Y} + Y) + X\bar{Z}(\bar{Y} + Y) \\
 &= \bar{X}\bar{Z} + X\bar{Z} = \bar{Z}(\bar{X} + X) = \bar{Z}
 \end{aligned}$$

The rectangle for this product term is shown in Figure 2-13(a). Note that the product term \bar{Z} uses the fact that the left and right edges of the map are adjacent in order to form the rectangle. Two other examples of rectangles corresponding to product terms derived from four minterms are shown in Figure 2-13(b).

In general, as more squares are combined, we obtain a product term with fewer literals. Three-variable maps exhibit the following characteristics:

One square represents a minterm of three literals.

A rectangle of two squares represents a product term of two literals.

A rectangle of four squares represents a product term of one literal.

A rectangle of eight squares encompasses the entire map and produces a function that is always equal to logic 1.

These characteristics are illustrated in Example 2-4.

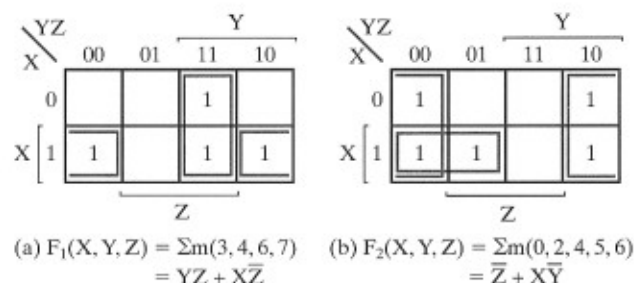
EXAMPLE 2-4 Simplifying Three-Variable Functions with Maps

Simplify the following two Boolean functions:

$$F_1(X, Y, Z) = \Sigma m(3, 4, 6, 7)$$

$$F_2(X, Y, Z) = \Sigma m(0, 2, 4, 5, 6)$$

The map for F_1 is shown in Figure 2-14(a). There are four squares marked with 1's, one for each minterm of the function. Two adjacent squares are combined



□ FIGURE 2-14

Maps for Example 2-4

in the third column to give a two-literal term YZ . The remaining two squares with 1's are also adjacent by the cylinder-based definition and are shown in the diagram with their values enclosed in half rectangles. When combined, these two squares give the two-literal term $X\bar{Z}$. The optimized function thus becomes

$$F_1 = YZ + X\bar{Z}$$

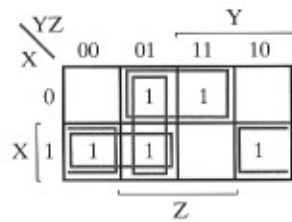
The map for F_2 is shown in Figure 2-14(b). First, we immediately combine the four adjacent squares in the first and last columns based on what we learned from Figure 2-13, to give the single literal term \bar{Z} . The remaining single square representing minterm 5 is combined with an adjacent square that already is being used once. This is not only permissible, but desirable, since the two adjacent squares give the two-literal term $X\bar{Y}$, while the single square represents the three-literal minterm $X\bar{Y}Z$. The optimized function is

$$F_2 = \bar{Z} + X\bar{Y}$$

On occasion there are alternative ways of combining squares to produce equally optimized expressions. An example of this is demonstrated in the map of Figure 2-15. Minterms 1 and 3 are combined to give the term $\bar{X}Z$, and minterms 4 and 6 produce the term $X\bar{Z}$. However, there are two ways that the square of minterm 5 can be combined with another adjacent square to produce a third two-literal term. Combining it with minterm 4 gives the term $X\bar{Y}$; combining it instead with minterm 1 gives the term $\bar{Y}Z$. Each of the two possible optimized expressions listed in Figure 2-15 has three terms of two literals each, so there are two possible optimized solutions for this function.

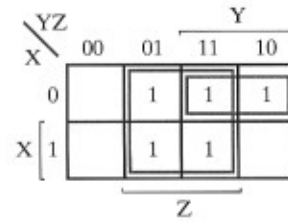
If a function is not expressed as a sum of minterms, we can use the map to obtain the minterms of the function and then simplify the function. It is necessary, however, to have the algebraic expression in sum-of-products form, from which each product term is plotted in the map. The minterms of the function are then read directly from the map. As an example, consider the Boolean function

$$F = \bar{X}Z + \bar{X}Y + X\bar{Y}Z + YZ$$



□ FIGURE 2-15

$$\begin{aligned}
 F(X, Y, Z) &= \Sigma m(1, 3, 4, 5, 6) \\
 &= \bar{X}Z + X\bar{Z} + X\bar{Y} \\
 &= \bar{X}Z + X\bar{Z} + \bar{Y}Z
 \end{aligned}$$



□ FIGURE 2-16

$$F(X, Y, Z) = \Sigma m(1, 2, 3, 5, 7) = Z + \bar{X}Y$$

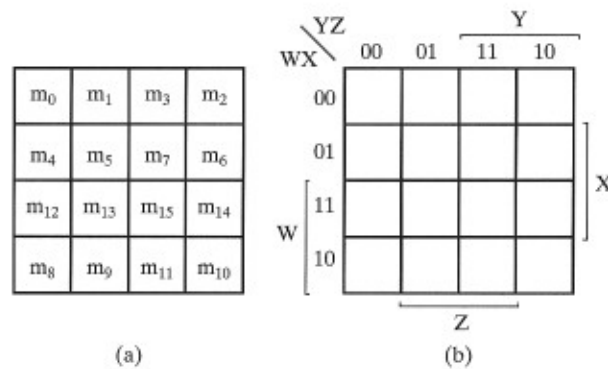
Three product terms in the expression have two literals and are represented in a three-variable map by two squares each. The two squares corresponding to the first term, $\bar{X}Z$, are found in Figure 2-16 from the coincidence of \bar{X} (first row) and Z (two middle columns), to give 1's in squares 001 and 011. Note that when marking 1's in the squares, it is possible to find a 1 already placed there from a preceding term. This happens with the second term, $\bar{X}Y$, which has 1's in squares 011 and 010; but square 011 is common with the first term, $\bar{X}Z$, so only one 1 is marked in it. Continuing in this fashion, we find that the function has five minterms, as indicated by the five 1's in the figure. The minterms are read directly from the map to be 1, 2, 3, 5, and 7. The function as originally given has four product terms. It can be optimized on the map to only two simple terms as

$$F = Z + \bar{X}Y$$

giving a significant reduction in the cost of implementation.

Four-Variable Map

There are 16 minterms for four binary variables, and therefore, a four-variable map consists of 16 squares, as shown in Figure 2-17. The minterm assignment in each square is indicated in part (a) of the diagram. The map is redrawn in (b) to show the relationship of the four variables. The rows and columns are numbered so that only one bit of the binary number changes in value between any two adjacent columns or rows, guaranteeing the same property for adjacent squares. The row and column number corresponds to a two-bit Gray code, as introduced in Chapter 1. The minterms corresponding to each square can be obtained by combining the row number with the column number. For example, when combined, the numbers in the third row (11) and the second column (01) give the binary number 1101, the binary equivalent of 13. Thus, the square in the third row and second column represents minterm m_{13} . In addition, each variable is marked on the map to show the eight squares in which it appears uncomplemented. The other eight squares, in which no label is indicated, correspond to the variable being complemented. Thus, W appears complemented in the first two rows and uncomplemented in the second two rows.



□ **FIGURE 2-17**
Four-Variable Map

The method used to simplify four-variable functions is similar to that used to simplify three-variable functions. Adjacent squares are defined to be squares next to each other, as for two- and three-variable maps. To show adjacencies between squares, the map of Figure 2-18(a) is drawn as a torus in Figure 2-18(b), with the top and bottom edges, as well as the right and left edges, touching each other to show adjacent squares. For example, m_0 and m_2 are two adjacent squares, as are m_0 and m_8 . The combinations of squares that can be chosen during the optimization process in the four-variable map are as follows:

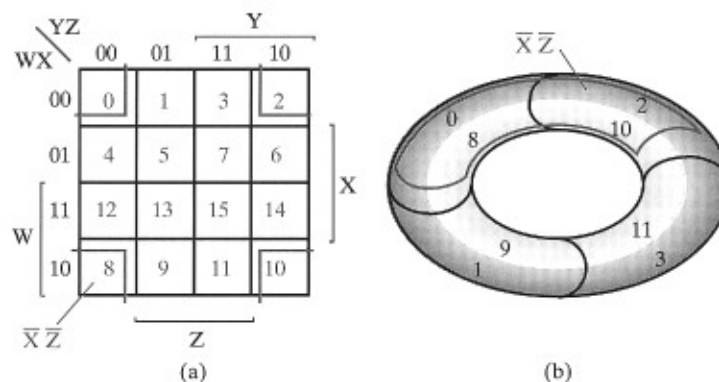
One square represents a minterm of four literals.

A rectangle of 2 squares represents a product term of three literals.

A rectangle of 4 squares represents a product term of two literals.

A rectangle of 8 squares represents a product term of one literal.

A rectangle of 16 squares produces a function that is always equal to logic 1.



□ **FIGURE 2-18**
Four-Variable Map: Flat and on a Torus to Show Adjacencies

No other combination of squares can be used. An interesting product term of two literals $X\bar{Z}$, is shown in Figure 2-18. In (b), when the map is viewed as a torus, the adjacencies of the squares that represent this product term are clear, but in (a) these squares are on the four corners of the map and appear quite removed from each other. This product term is important to recall, since it is often missed. It also serves as a reminder that the left edge and the right edge of the map are adjacent, as are the top edge and the bottom edge. Thus, in general, rectangles on a map cross the left and right edges, top and bottom edges, or both.

The next examples show the procedure for simplifying four-variable Boolean functions.

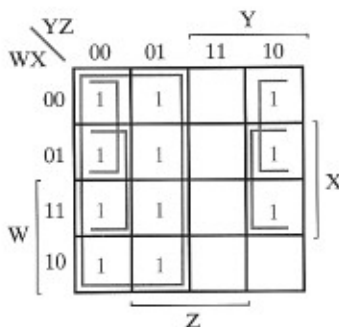
EXAMPLE 2-5 Simplifying a 4-Variable Function with a Map

Simplify the Boolean function

$$F(W, X, Y, Z) = \sum m(0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14)$$

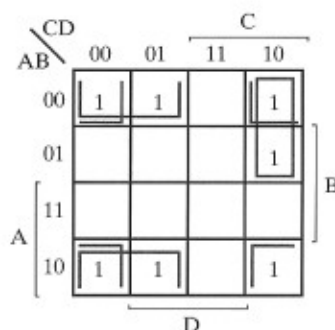
The minterms of the function are marked with 1's in the map of Figure 2-19. Eight squares in the two left columns are combined to form a rectangle for the one literal term, \bar{Y} . The remaining three 1's cannot be combined to give a simplified term; rather, they must be combined as two- or four-square rectangles. The top two 1's on the right are combined with the top two 1's on the left to give the term $\bar{W}\bar{Z}$. Note again that it is permissible to use the same square more than once. We are now left with a square marked with a 1 in the third row and fourth column (minterm 1110). Instead of taking this square alone, which will give a term of four literals, we combine it with squares already used to form a rectangle of four squares in the two middle rows and the two end columns, giving the term $X\bar{Z}$. The optimized expression is the logical sum of the three terms:

$$F = \bar{Y} + \bar{W}\bar{Z} + X\bar{Z}$$



□ FIGURE 2-19

Map for Example 2-5: $F = \bar{Y} + \bar{W}\bar{Z} + X\bar{Z}$



□ FIGURE 2-20

Map for Example 2-6: $F = \overline{B}\overline{D} + \overline{B}\overline{C} + \overline{A}C\overline{D}$ **EXAMPLE 2-6 Simplifying a 4-Variable Function with a Map**

Simplify the Boolean function

$$F = \overline{A}\overline{B}\overline{C} + \overline{B}C\overline{D} + A\overline{B}\overline{C} + \overline{A}BC\overline{D}$$

This function has four variables: A , B , C , and D . It is expressed in sum-of-products form with three terms of three literals each and one term of four literals. The area in the map covered by the function is shown in Figure 2-20. Each term of three literals is represented in the map by two squares. $\overline{A}\overline{B}\overline{C}$ is represented by squares 0000 and 0001, $\overline{B}C\overline{D}$ by squares 0010 and 1010, and $A\overline{B}\overline{C}$ by squares 1000 and 1001. The term with four literals is minterm 0110. The function is simplified on the map by taking the 1's in the four corners, to give the term $\overline{B}\overline{D}$. This product term is in the same map locations as $\overline{X}\overline{Z}$ in Figure 2-18. The two 1's in the top row are combined with the two 1's in the bottom row to give the term $\overline{B}\overline{C}$. The remaining 1, in square 0110, is combined with its adjacent square, 0010, to give the term $\overline{A}C\overline{D}$. The optimized function is thus

$$F = \overline{B}\overline{D} + \overline{B}\overline{C} + \overline{A}C\overline{D}$$

■

2-5 MAP MANIPULATION

When combining squares in a map, it is necessary to ensure that all the minterms of the function are included. At the same time, it is necessary to minimize the number of terms in the optimized function by avoiding any redundant terms whose minterms are already included in other terms. In this section, we consider a procedure that assists in the recognition of useful patterns in the map. Other topics to be covered are the optimization of products of sums and the optimization of incompletely specified functions.

Essential Prime Implicants

The procedure for combining squares in a map may be made more systematic if we introduce the terms “implicant,” “prime implicant,” and “essential prime implicant.” A product term is an *implicant* of a function if the function has the value 1 for all minterms of the product term. Clearly, all rectangles on a map made up of squares containing 1's correspond to implicants. If the removal of any literal from an implicant P results in a product term that is not an implicant of the function, then P is a *prime implicant*. On a map for an n -variable function, the set of prime implicants corresponds to the set of all rectangles made up of 2^m squares containing 1's ($m = 0, 1, \dots, n$), with each rectangle containing as many squares as possible.

If a minterm of a function is included in only one prime implicant, that prime implicant is said to be *essential*. Thus, if a square containing a 1 is in only one rectangle representing a prime implicant, then that prime implicant is essential. In Figure 2-15 on page 55, the terms $\overline{X}Z$ and $X\overline{Z}$ are essential prime implicants, and the terms $\overline{X}Y$ and $\overline{Y}Z$ are nonessential prime implicants.

The prime implicants of a function can be obtained from a map of the function as all possible maximum collections of 2^m squares containing 1's ($m = 0, 1, \dots, n$) that constitute rectangles. This means that a single 1 on a map represents a prime implicant if it is not adjacent to any other 1's. Two adjacent 1's form a rectangle representing a prime implicant, provided that they are not within a rectangle of four or more squares containing 1's. Four 1's form a rectangle representing a prime implicant if they are not within a rectangle of eight or more squares containing 1's, and so on. Each essential prime implicant contains at least one square that is not contained in any other prime implicant.

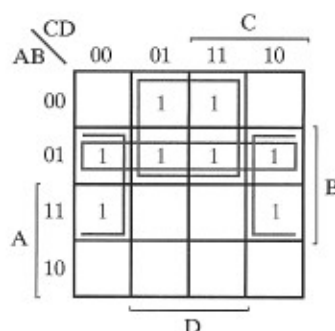
The systematic procedure for finding the optimized expression from the map requires that we first determine all prime implicants. Then, the optimized expression is obtained from the logical sum of all the essential prime implicants, plus other prime implicants needed to include remaining minterms not included in the essential prime implicants. This procedure will be clarified by examples.

EXAMPLE 2-7 Simplification Using Prime Implicants

Consider the map of Figure 2-21. There are three ways that we can combine four squares into rectangles. The product terms obtained from these combinations are the prime implicants of the function, $\overline{A}D$, $B\overline{D}$ and $\overline{A}B$. The terms $\overline{A}D$ and $B\overline{D}$ are essential prime implicants, but $\overline{A}B$ is not essential. This is because minterms 1 and 3 can be included only in the term $\overline{A}D$, and minterms 12 and 14 can be included only in the term $B\overline{D}$. But minterms 4, 5, 6, and 7 are each included in two prime implicants, one of which is $\overline{A}B$, so the term $\overline{A}B$ is not an essential prime implicant. In fact, once the essential prime implicants are chosen, the third term is not needed because all the minterms are already included in the essential prime implicants. The optimized expression for the function of Figure 2-21 is

$$F = \overline{A}D + B\overline{D}$$



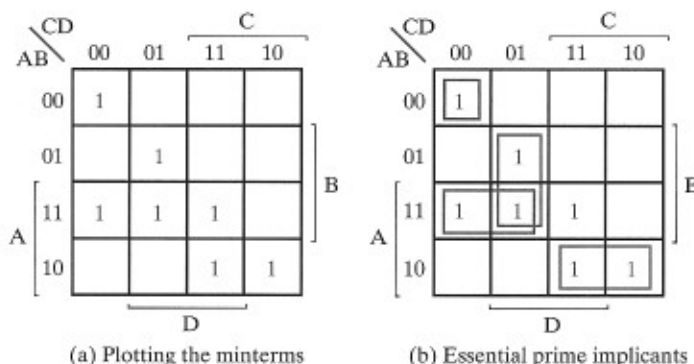


□ **FIGURE 2-21**
Prime Implicants for Example 2-7: $\overline{A}D$, $B\overline{D}$, and $\overline{A}B$

EXAMPLE 2-8 Simplification Via Essential and Nonessential Prime Implicants

A second example is shown in Figure 2-22. The function plotted in part (a) has seven minterms. If we try to combine squares, we will find that there are six prime implicants. In order to obtain a minimum number of terms for the function, we must first determine the prime implicants that are essential. As shown in part (b) of the figure, the function has four essential prime implicants. The product term $\overline{A}\overline{B}\overline{C}\overline{D}$ is essential because it is the only prime implicant that includes minterm 0. Similarly, the product terms $B\overline{C}D$, $AB\overline{C}$, and $A\overline{B}C$ are essential prime implicants because they are the only prime implicants that include minterms 5, 12, and 10, respectively. Minterm 15 is included in two nonessential prime implicants. The optimized expression for the function consists of the logical sum of the four essential prime implicants and one prime implicant that includes minterm 15:

$$F = \overline{A}\overline{B}\overline{C}\overline{D} + B\overline{C}D + AB\overline{C} + A\overline{B}C + \begin{cases} ACD \\ \text{or} \\ ABD \end{cases}$$



□ **FIGURE 2-22**
Simplification with Prime Implicants in Example 2-8

The identification of essential prime implicants in the map provides an additional tool which shows the terms that must absolutely appear in every sum-of-products expression for a function and provides a partial structure for a more systematic method for choosing patterns of squares.

Nonessential Prime Implicants

Beyond using all essential prime implicants, the following rule can be applied to include the remaining minterms of the function in nonessential prime implicants:

Selection Rule: Minimize the overlap among prime implicants as much as possible. In particular, in the final solution, make sure that each prime implicant selected includes at least one minterm not included in any other prime implicant selected.

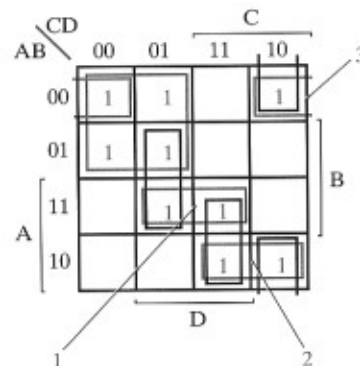
In most cases, this results in a simplified, although not necessarily minimum cost, sum-of-products expression. The use of the selection rule is illustrated in the next example.

EXAMPLE 2-9 Simplifying a Function Using the Selection Rule

Find a simplified sum-of-products form for $F(A, B, C, D) = \sum m(0, 1, 2, 4, 5, 10, 11, 13, 15)$.

The map for F is given in Figure 2-23, with all prime implicants shown. $\overline{A}\overline{C}$ is the only essential prime implicant. Using the preceding selection rule, we can choose the remaining prime implicants for the sum-of-products form in the order indicated by the numbers. Note how the prime implicants 1 and 2 are selected in order to include minterms without overlapping. Prime implicant 3 ($\overline{A}\overline{B}\overline{D}$) and prime implicant $\overline{B}\overline{C}\overline{D}$ both include the one remaining minterm 0010, and prime implicant 3 is arbitrarily selected to include the minterm and complete the sum-of-products expression:

$$F(A, B, C, D) = \overline{A}\overline{C} + ABD + \overline{A}\overline{B}\overline{C} + \overline{A}\overline{B}\overline{D}$$



□ FIGURE 2-23
Map for Example 2-9

Product-of-Sums Optimization

The optimized Boolean functions derived from the maps in all of the previous examples were expressed in sum-of-products form. With only minor modification, the product-of-sums form can be obtained.

The procedure for obtaining an optimized expression in product-of-sums form follows from the basic properties of Boolean functions. The 1's placed in the squares of the map represent the minterms of the function. The minterms not included in the function belong to the complement of the function. From this, we see that the complement of a function is represented in the map by the squares not marked by 1's. If we mark the empty squares with 0's and combine them into valid rectangles, we obtain an optimized expression of the complement of the function. We then take the complement of \bar{F} to obtain the function F as a product of sums. This is done by taking the dual and complementing each literal, as described in Example 2-2 on page 41.

EXAMPLE 2-10 Simplifying a Product-of-Sums Form

Simplify the following Boolean function in product-of-sums form:

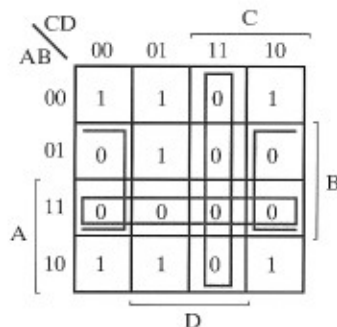
$$F(A, B, C, D) = \Sigma m(0, 1, 2, 5, 8, 9, 10)$$

The 1's marked in the map of Figure 2-24 represent the minterms of the function. The squares marked with 0's represent the minterms not included in F and therefore denote the complement of F . Combining the squares marked with 0's, we obtain the optimized complemented function

$$\bar{F} = AB + CD + B\bar{D}$$

Taking the dual and complementing each literal gives the complement of \bar{F} . This is F in product-of-sums form:

$$F = (\bar{A} + \bar{B})(\bar{C} + \bar{D})(\bar{B} + D)$$



□ FIGURE 2-24

Map for Example 2-10: $F = (\bar{A} + \bar{B})(\bar{C} + \bar{D})(\bar{B} + D)$

The previous example shows the procedure for obtaining the product-of-sums optimization when the function is originally expressed as a sum of minterms. The procedure is also valid when the function is originally expressed as a product of maxterms or a product of sums. Remember that the maxterm numbers are the same as the minterm numbers of the complemented function, so 0's are entered in the map for the maxterms or for the complement of the function. To enter a function expressed as a product of sums into the map, we take the complement of the function and, from it, find the squares to be marked with 0's. For example, the function

$$F = (\bar{A} + \bar{B} + C)(B + D)$$

can be plotted in the map by first obtaining its complement,

$$\bar{F} = ABC + \bar{B}\bar{D}$$

and then marking 0's in the squares representing the minterms of \bar{F} . The remaining squares are marked with 1's. Then, combining the 1's gives the optimized expression in sum-of-products form. Combining the 0's and then complementing gives the optimized expression in product-of-sums form. Thus, for any function plotted on the map, we can derive the optimized function in either one of the two standard forms.

Don't-Care Conditions

The minterms of a Boolean function specify all combinations of variable values for which the function is equal to 1. The function is assumed to be equal to 0 for the rest of the minterms. This assumption, however, is not always valid, since there are applications in which the function is not specified for certain variable value combinations. There are two cases in which this occurs. In the first case, the input combinations never occur. As an example, the four-bit binary code for the decimal digits has six combinations that are not used and not expected to occur. In the second case, the input combinations are expected to occur, but we do not care what the outputs are in response to these combinations. In both cases, the outputs are said to be unspecified for the input combinations. Functions that have unspecified outputs for some input combinations are called *incompletely specified functions*. In most applications, we simply do not care what value is assumed by the function for the unspecified minterms. For this reason, it is customary to call the unspecified minterms of a function *don't-care conditions*. These conditions can be used on a map to provide further simplification of the function.

It should be realized that a don't-care minterm cannot be marked with a 1 on the map, because that would require that the function always be a 1 for such a minterm. Likewise, putting a 0 in the square requires the function to be 0. To distinguish the don't-care condition from 1's and 0's, an X is used. Thus, an X inside a square in the map indicates that we do not care whether the value of 0 or 1 is assigned to the function for the particular minterm.

In choosing adjacent squares to simplify the function in a map, the don't-care minterms may be used. When simplifying function F , using the 1's we can choose to include those don't-care minterms that give the simplest prime implicants for F . When simplifying function \bar{F} using the 0's, we can choose to include those don't-care minterms that give the simplest prime implicants for \bar{F} , irrespective of those included in the prime implicants for F . In both cases, whether or not the don't care minterms are included in the terms in the final expression is irrelevant. The handling of don't care conditions is illustrated in the next example.

EXAMPLE 2-11 Simplification with Don't Care Conditions

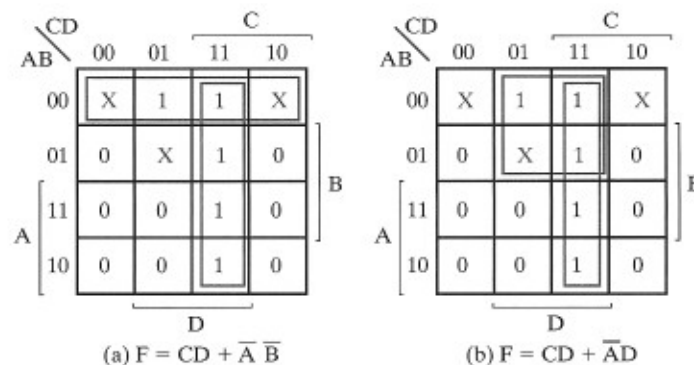
To clarify the procedure for handling the don't-care conditions, consider the following incompletely specified function F that has three don't-care minterms d :

$$F(A, B, C, D) = \sum m(1, 3, 7, 11, 15)$$

$$d(A, B, C, D) = \sum m(0, 2, 5)$$

The minterms of F are the variable combinations that make the function equal to 1. The minterms of d are the don't-care minterms. The map optimization is shown in Figure 2-25. The minterms of F are marked by 1's, those of d are marked by X's, and the remaining squares are filled with 0's. To get the simplified function in sum-of-products form, we must include all five 1's in the map, but we may or may not include any of the X's, depending on what yields the simplest expression for the function. The term CD includes the four minterms in the third column. The remaining minterm in square 0001 can be combined with square 0011 to give a three-literal term. However, by including one or two adjacent X's, we can combine four squares into a rectangle to give a two-literal term. In part (a) of the figure, don't-care minterms 0 and 2 are included with the 1's, which results in the simplified function

$$F = CD + \bar{A}\bar{B}$$



□ **FIGURE 2-25**
Example with Don't-Care Conditions

In part (b), don't-care minterm 5 is included with the 1's, and the simplified function now is

$$F = CD + \bar{A}D$$

The two expressions represent two functions that are algebraically unequal. Both include the specified minterms of the original incompletely specified function, but each includes different don't-care minterms. As far as the incompletely specified function is concerned, both expressions are acceptable. The only difference is in the value of F for the unspecified minterms.

It is also possible to obtain an optimized product-of-sums expression for the function of Figure 2-25. In this case, the way to combine the 0's is to include don't-care minterms 0 and 2 with the 0's giving the optimized complemented function

$$\bar{F} = \bar{D} + A\bar{C}$$

Taking the complement of \bar{F} gives the optimized expression in product-of-sums form:

$$F = D(\bar{A} + C)$$

The foregoing example shows that the don't-care minterms in the map are initially considered as representing both 0 and 1. The 0 or 1 value eventually assigned depends on the optimization process. Due to this process, the optimized function will have a 0 or 1 value for each minterm of the original function, including those that were initially don't cares. Thus, although the outputs in the initial specification may contain X's, the outputs in a particular implementation of the specification are only 0's and 1's.

2-6 MULTIPLE-LEVEL CIRCUIT OPTIMIZATION

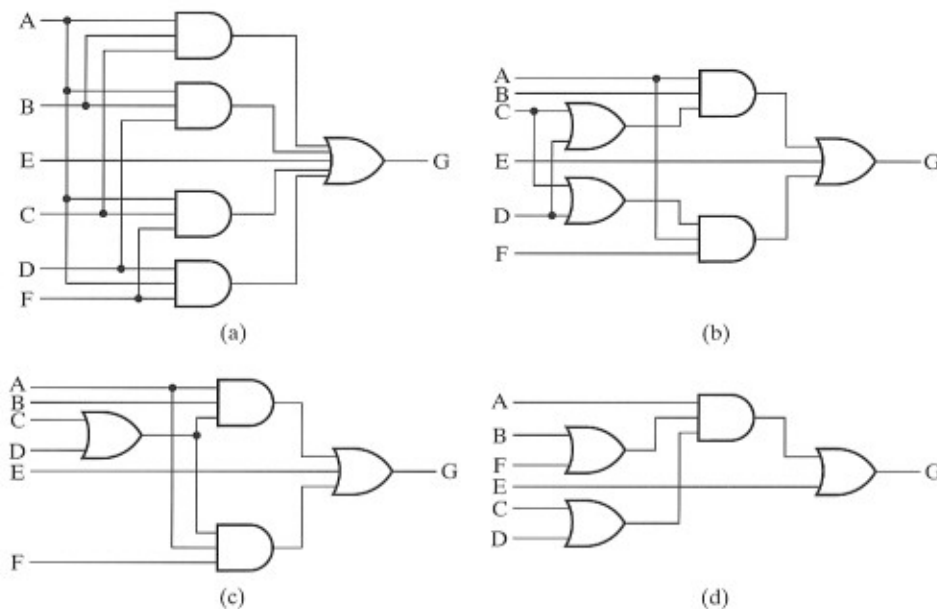
Although we have found that two-level circuit optimization can reduce the cost of combinational logic circuits, there are often additional cost savings available by using circuits with more than two levels. Such circuits are referred to as multiple-level circuits. These savings are illustrated by the implementation of the function

$$G = ABC + ABD + E + ACF + ADF$$

Figure 2-26(a) gives the two-level implementation of G which has a gate-input cost of 17. Now suppose that we apply the distributive law of Boolean algebra to G to give

$$G = AB(C + D) + E + A(C + D)F$$

This equation gives the multiple-level implementation of G in Figure 2-26(b) which has a gate input cost of 13, an improvement of 4 gate inputs. In Figure 2-26(b),



□ **FIGURE 2-26**
Multiple-Level Circuit Example

$C + D$ is implemented twice. Instead, one implementation of this subfunction can be shared to give the circuit in Figure 2-26(c) with a gate-input cost of 11, an improvement of 2. This common use of $(C + D)$ suggests that G can be written as

$$G = (AB + AF)(C + D) + E$$

This increases the cost to 12. But by factoring out A from $AB + AF$, we obtain

$$G = A(B + F)(C + D) + E$$

Figure 2-26(d) gives the multi-level implementation of G using this equation which has a gate input cost of only nine, is slightly more than one-half of the original cost.

This reduction was achieved by a sequence of applications of algebraic identities at each step observing the effect on the gate input cost. Just as with the use of Boolean algebra to obtain simplified two-level circuits, the procedure used here is not particularly systematic. Further, an algorithmic procedure corresponding to the use of Karnaugh maps for two-level circuit optimization that gives an optimum circuit cost does not exist due to the broader range of possible actions and the number of solutions possible. So multiple-level optimization is based on the use of a set of transformations that are applied in conjunction with cost evaluation

to find a good, but not necessarily optimum solution. In the remainder of this section, we consider such transformations and illustrate their application in reducing circuit cost. The transformations, to be illustrated by the next example, are defined as follows:

1. *Factoring* is finding a factored form from either a sum-of-products expression or product-of-sums expression for a function.
2. *Decomposition* is the expression of a function as a set of new functions.
3. *Extraction* is the expression of multiple functions as a set of new functions.
4. *Substitution* of a function G into a function F is expressing F as a function of G and some or all of the original variables of F .
5. *Elimination* is the inverse of substitution in which function G in an expression for function F is replaced by the expression for G . Elimination is also called *flattening* or *collapsing*.

EXAMPLE 2-12 Multilevel Optimization Transformations

The following functions will be used in illustrating the transformations:

$$\begin{aligned} G &= A \bar{C} E + A \bar{C} F + A \bar{D} E + A \bar{D} F + B C D \bar{E} \bar{F} \\ H &= \bar{A} B C D + A C E + A C F + B C E + B C F \end{aligned}$$

The first transformation to be illustrated is factoring by using function G . Initially, we will look at *algebraic factoring*, which avoids axioms that are unique to Boolean algebra, such as those involving the complement and idempotence. Factors can be found not only for the entire expression for G , but also for its subexpressions. For example, since the first four terms of G all contain variable A , it can be factored out of these terms giving:

$$G = A (\bar{C} E + \bar{C} F + \bar{D} E + \bar{D} F) + B C D \bar{E} \bar{F}$$

In this case, note that A and $\bar{C} E + \bar{C} F + \bar{D} E + \bar{D} F$ are factors, and $B C D \bar{E} \bar{F}$ is not involved in the factoring operation. By factoring out \bar{C} and \bar{D} , $\bar{C} E + \bar{C} F + \bar{D} E + \bar{D} F$ can be written as $\bar{C} (E + F) + \bar{D} (E + F)$ which be rewritten as $(\bar{C} + \bar{D})(E + F)$. Placing this expression in G gives:

$$G = A(\bar{C} + \bar{D})(E + F) + B C D \bar{E} \bar{F}$$

The term $B C D \bar{E} \bar{F}$ could be factored into product terms, but such factoring will not reduce the gate input count and so is not considered. The gate input count for the original sum-of-products expression for G is 26 and for the factored form of G is 18, for a saving of 8 gate inputs. Due to the factoring, there are more gates in series from inputs to outputs, a maximum of four levels instead of three levels including input inverters. This may result in an increase in the delay through the circuit after technology mapping has been applied.

The second transformation to be illustrated is decomposition which allows operations beyond algebraic factoring. The factored form of G can be written as a decomposition as follows:

$$\begin{aligned}G &= A (\bar{C} + \bar{D}) X_2 + B X_1 \bar{E} \bar{F} \\X_1 &= C D \\X_2 &= E + F\end{aligned}$$

Once X_1 and X_2 have been defined, they can be complemented, and the complements can replace $\bar{C} + \bar{D}$ and $\bar{E} \bar{F}$, respectively, in G . An illustration of the substitution transformation is

$$\begin{aligned}G &= A \bar{X}_1 X_2 + B X_1 \bar{X}_2 \\X_1 &= C D \\X_2 &= E + F\end{aligned}$$

The gate input count for this decomposition is 14, for a saving of 12 gate inputs from the original sum-of-products expression for G , and of 4 gate inputs from the factored form of G .

In order to illustrate extraction, we need to perform decomposition on H and extract common subexpressions in G and H . Factoring out B from H , we have

$$H = B(\bar{A} C D + A E + A + C E + C F)$$

Determining additional factors in H , we can write

$$H = B(\bar{A} (C D) + (A + C)(E + F))$$

Factors X_1 , X_2 , and X_3 can now be extracted to obtain

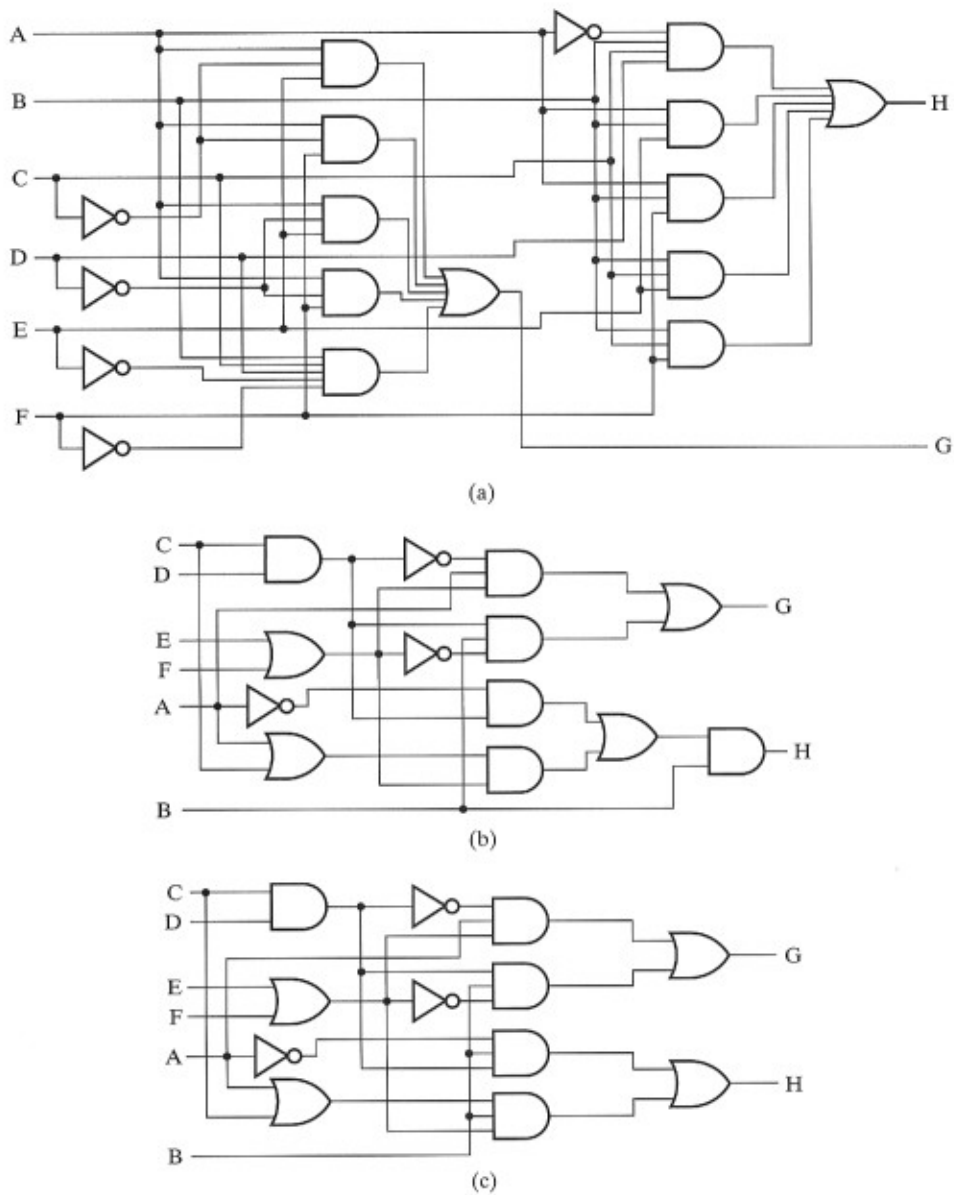
$$\begin{aligned}X_1 &= C D \\X_2 &= E + F \\X_3 &= A + C\end{aligned}$$

and factors X_1 and X_2 can be shared between G and H . Performing substitution, we can write G and H as

$$\begin{aligned}G &= A \bar{X}_1 X_2 + B X_1 \bar{X}_2 \\H &= B(\bar{A} X_1 + X_3 X_2)\end{aligned}$$

A logic diagram is given for the original sum-of-products in Figure 2-27(a) and for the extracted form in Figure 2-27(b). The gate input cost for the original G and H without shared terms, except for input inverters, is 48. For decomposed G and H without shared terms between G and H , it is 31. With shared terms, it is 25, cutting the gate input cost in half. ■

This example illustrates the value of the transformations in reducing input count cost. In general, due to the wide range of alternative solutions and the complexity in



□ **FIGURE 2-27**
Multiple-Level Circuit Optimization Example

determining the divisors to use in decomposition and extraction, obtaining truly optimum solutions in terms of gate input count is usually not feasible, so only good solutions are sought. The key to successful transformations is the determination of the

factors to be used in decomposition or extraction and choice of the transformation sequence to apply. These decisions are complex and beyond the scope of our study here, but are regularly incorporated into logic synthesis tools.

Our discussion thus far has dealt only with multilevel optimization in terms of reducing gate input count. In a large proportion of designs, the length of the longest path or paths through the circuit is often constrained due to the path delay, the length of time it takes for a change in a signal to propagate down a path through the gates. In such cases, the number of gates in series may need to be reduced. Such a reduction using the final transformation, elimination, is illustrated in the following example.

EXAMPLE 2-13 Example of Transformation for Delay Reduction

In the circuit in Figure 2-27(b), the paths from C , D , E , F and A to H all pass through four 2-input gates. Assuming that all multi-input gates contribute the same delay to the path, a delay greater than that contributed by an inverter, these are the longest delay paths in the circuit. Due to a specification on maximum path delay for the circuit, these paths must be shortened to at most three multi-input gates or their equivalent in multi-input gates and inverter delays. This path shortening should be done with a minimum increase in gate input count.

The elimination transform which replaces intermediate variables, X_i , with the expressions on their right hand sides or removes other factoring such as that of variable B is the mechanism for reducing the number of gates in series. To determine which factor or combination of factors should be eliminated, we need to look at the effect on gate input count. The increase in gate input count for the combinations of eliminations that reduce the problem path lengths by at least one gate are of interest. There are only three such combinations: elimination of the factoring of B , elimination of intermediate variables X_1 , X_2 , and X_3 , and elimination of the factor B and the three intermediate variables X_1 , X_2 , and X_3 . The respective gate input count increases for these actions are 0, 12, and 12, respectively. Clearly, the removal of the factor B is the best choice since the gate input count does not increase. This also demonstrates that, due to the additional decomposition of H , the gate input cost gain of 3 that occurred by factoring out B at the beginning has disappeared. The logic diagram resulting from elimination of the factor B is given in Figure 2-27(c). ■

While the necessary delay reduction was obtained by using elimination to reduce the number of gates along the paths in Example 2-13, in general, such a gate reduction may not reduce delay, or may even increase it due to differences in the delay characteristics of the gates to be discussed further in Chapter 3.

2-7 OTHER GATE TYPES

Since Boolean functions are expressed in terms of AND, OR, and NOT operations, it is a straightforward procedure to implement a Boolean function with AND, OR,



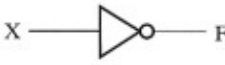
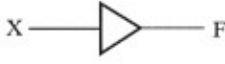
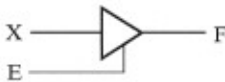


and NOT gates. We find, however, that the possibility of considering gates with other logic operations is of considerable practical interest. Factors to be taken into consideration when constructing other types of gates are the feasibility and economy of implementing the gate with electronic components, the ability of the gate to implement Boolean functions alone or in conjunction with other gates, and the convenience of representing gate functions that are frequently used. In this section, we introduce these other gate types which are used throughout the rest of the text. Specific techniques for incorporating these gate types in circuits are given in section 3-5.

The graphics symbols and truth tables of six logic gate types are shown in Figure 2-28, with six additional gate types given in Figure 2-29. The gates in Figure 2-28 are referred to as *primitive* gates, and those in Figure 2-29 are referred to as *complex* gates.



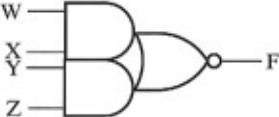
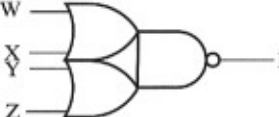
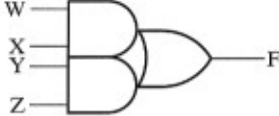
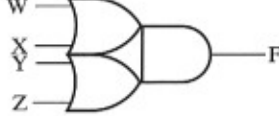
Although the gates in Figure 2-28 are shown with just two binary input variables, X and Y , and one output binary variable, F , with the exception of the inverter and the buffer, all may have more than two inputs. The distinctively shaped symbols shown, as well as rectangular symbols not shown, are specified in detail in the Institute of Electrical and Electronics Engineers' (IEEE) *Standard Graphic Symbols for Logic Functions* (IEEE Standard 91-1984). The AND, OR, and NOT gates were defined previously. The NOT circuit inverts the logic sense of a binary signal to produce the complement operation. Recall that this circuit is typically called an *inverter* rather than a NOT gate. The small circle at the output of the graphic symbol of an inverter is formally called a *negation indicator* and designates the logical complement. We informally refer to the negation indicator as a "bubble." The triangle symbol by itself designates a buffer circuit. A *buffer* produces the logical function $Z = X$, since the binary value of the output is equal to the binary value of the input. This circuit is used primarily to amplify an electrical signal to permit more gates to be attached to the output or to decrease the time it takes for signals to propagate through the circuit.

The 3-state buffer is unique in that outputs of 3-state buffers can be connected together provided that only one of the signals on their E inputs is 1 at any given time. This type of buffer and its basic use are discussed in detail later in this section.

The NAND gate represents the complement of the AND operation, and the NOR gate represents the complement of the OR operation. Their respective names are abbreviations of NOT-AND and NOT-OR, respectively. The graphics symbols for the NAND gate and NOR gate consist of an AND symbol and an OR symbol, respectively, with a bubble on the output, denoting the complement operation. In contemporary integrated circuit technology, NAND and NOR gates are the natural primitive gate functions for the simplest and fastest electronic circuits. If we consider the inverter as a degenerate version of NAND and NOR gates with just one input, NAND gates alone or NOR gates alone can implement any Boolean function. Thus, these gate types are much more widely used than AND and OR gates in actual logic circuits. As a consequence, actual circuit implementations are often done in terms of these gate types.

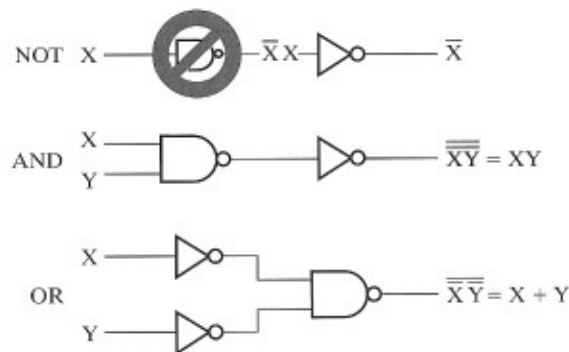
Graphics Symbols			
Name	Distinctive shape	Algebraic equation	Truth table
AND		$F = XY$	$X \ Y \ \ F$
			$0 \ 0 \ \ 0$
			$0 \ 1 \ \ 0$
			$1 \ 0 \ \ 0$
			$1 \ 1 \ \ 1$
OR		$F = X + Y$	$X \ Y \ \ F$
			$0 \ 0 \ \ 0$
			$0 \ 1 \ \ 1$
			$1 \ 0 \ \ 1$
			$1 \ 1 \ \ 1$
NOT (inverter)		$F = \overline{X}$	$X \ \ F$
			$0 \ \ 1$
			$1 \ \ 0$
Buffer		$F = X$	$X \ \ F$
			$0 \ \ 0$
			$1 \ \ 1$
3-State Buffer			$E \ X \ \ F$
			$0 \ 0 \ \ \text{Hi-Z}$
			$0 \ 1 \ \ \text{Hi-Z}$
			$1 \ 0 \ \ 0$
			$1 \ 1 \ \ 1$
NAND		$F = \overline{X \cdot Y}$	$X \ Y \ \ F$
			$0 \ 0 \ \ 1$
			$0 \ 1 \ \ 1$
			$1 \ 0 \ \ 1$
			$1 \ 1 \ \ 0$
NOR		$F = \overline{X + Y}$	$X \ Y \ \ F$
			$0 \ 0 \ \ 1$
			$0 \ 1 \ \ 0$
			$1 \ 0 \ \ 0$
			$1 \ 1 \ \ 0$

□ **FIGURE 2-28**
Primitive Digital Logic Gates

Graphics Symbols			
Name	Distinctive shape symbol	Algebraic equation	Truth table
Exclusive-OR (XOR)		$F = X\bar{Y} + \bar{X}Y$ $= X \oplus Y$	X Y F
			0 0 0
			0 1 1
			1 0 1
			1 1 0
Exclusive-NOR (XNOR)		$F = XY + \bar{X}\bar{Y}$ $= \overline{X \oplus Y}$	X Y F
			0 0 1
			0 1 0
			1 0 0
			1 1 1
AND-OR-INVERT (AOI)		$F = \overline{WX + YZ}$	W X Y Z F
			0 0 0 0 1
			0 0 1 0 1
			0 1 0 0 1
			0 1 1 0 1
OR-AND-INVERT (OAI)		$F = \overline{(W + X)(Y + Z)}$	W X Y Z F
			0 0 0 0 1
			0 0 1 0 1
			0 1 0 0 1
			0 1 1 0 1
AND-OR (AO)		$F = WX + YZ$	W X Y Z F
			0 0 0 0 0
			0 0 1 0 0
			0 1 0 0 0
			0 1 1 0 0
OR-AND (OA)		$F = (W + X)(Y + Z)$	W X Y Z F
			0 0 0 0 0
			0 0 1 0 0
			0 1 0 0 0
			0 1 1 0 0

□ **FIGURE 2-29**
Complex Digital Logic Gates

A gate type that alone can be used to implement all Boolean functions is called a *universal gate*. To show the NAND gate is a universal gate, we need only show that the logical operations of AND, OR, and NOT can be obtained with NAND gates only. This is done in Figure 2-30. The complement operation is obtained from a one-input NAND gate corresponds to a NOT gate. In fact, the one-input NAND is an invalid symbol and is replaced by the NOT symbol, as shown in the figure. The AND operation requires a NAND gate followed by a NOT gate. The NOT inverts the output of the NAND giving an AND operation as



□ **FIGURE 2-30**
Logical Operations with NAND Gates

the result. The OR operation is achieved using a NAND gate with NOTs on each input. When DeMorgan's theorem is applied as shown in Figure 2-30, the inversions cancel and an OR function results.

The exclusive-OR (XOR) gate shown in Figure 2-29 is similar to the OR gate, but excludes (has the value 0 for) the combination with both X and Y equal to 1. The graphics symbol for the XOR gate is similar to that for the OR gate, except for the additional curved line on the inputs. The exclusive-OR has the special symbol \oplus to designate its operation. The exclusive-NOR is the complement of the exclusive-OR, as indicated by the bubble at the output of its graphics symbol.

The AND-OR-INVERT (AOI) gate forms the complement of a sum-of-products. There are many different AND-OR-INVERT gates depending on the number of AND gates and the numbers of inputs to each AND and directly to the OR gate. For example, suppose that the function implemented by an AOI is

$$F = \overline{XY + Z}$$

This AOI is referred to as a 2-1 AOI since it consists of a 2-input AND and 1 input directly to the OR gate. If the function implemented is

$$F = \overline{TUV + WX + YZ}$$

then the AOI is called a 3-2-2 AOI. The OR-AND-INVERT (OAI) is the dual of the AOI and implements the complement of a product-of-sums form. The AND-OR (AO) and OR-AND (OA) are versions of the AOI and OAI without the complement.

In general, complex gates are used to reduce the circuit complexity needed for implementing specific Boolean functions in order to reduce integrated circuit cost. In addition, they reduce the time required for signals to propagate through a circuit.



CMOS CIRCUITS This supplement, which discusses the implementation of both primitive and complex gates with CMOS technology, is available on the Companion Website for the text.

2-8 EXCLUSIVE-OR OPERATOR AND GATES

In addition to the exclusive-OR gate shown in Figure 2-29, there is an exclusive-OR operator with its own algebraic identities. The exclusive-OR (XOR), denoted by \oplus , is a logical operation that performs the function

$$X \oplus Y = X\bar{Y} + \bar{X}Y$$

It is equal to 1 if exactly one input variable is equal to 1. The exclusive-NOR, also known as the *equivalence*, is the complement of the exclusive-OR and is expressed by the function

$$\overline{X \oplus Y} = XY + \bar{X}\bar{Y}$$

It is equal to 1 if both X and Y are equal to 1 or if both are equal to 0. The two functions can be shown to be the complement of each other, either by means of a truth table or, as follows, by algebraic manipulation:

$$\overline{X \oplus Y} = \overline{X\bar{Y} + \bar{X}Y} = (\bar{X} + Y)(X + \bar{Y}) = XY + \bar{X}\bar{Y}$$

The following identities apply to the exclusive-OR operation:

$$\begin{aligned} X \oplus 0 &= X & X \oplus 1 &= \bar{X} \\ X \oplus X &= 0 & X \oplus \bar{X} &= 1 \\ X \oplus \bar{Y} &= \overline{X \oplus Y} & \bar{X} \oplus Y &= \overline{X \oplus Y} \end{aligned}$$

Any of these identities can be verified by using a truth table or by replacing the \oplus operation by its equivalent Boolean expression. It can also be shown that the exclusive-OR operation is both commutative and associative; that is,

$$\begin{aligned} A \oplus B &= B \oplus A \\ (A \oplus B) \oplus C &= A \oplus (B \oplus C) = A \oplus B \oplus C \end{aligned}$$

This means that the two inputs to an exclusive-OR gate can be interchanged without affecting the operation. It also means that we can evaluate a three-variable exclusive-OR operation in any order, and for this reason, exclusive-ORs with three or more variables can be expressed without parentheses.

A two-input exclusive-OR function may be constructed with conventional gates. Two NOT gates, two AND gates, and an OR gate are used. The associativity of the exclusive-OR operator suggests the possibility of exclusive-OR gates with more than two inputs. The exclusive-OR concept for more than two variables, however, is replaced by the odd function to be discussed next. Thus, there is no symbol for exclusive-OR for more than two inputs. By duality, the exclusive-NOR is replaced by the even function and has no symbol for more than two inputs.

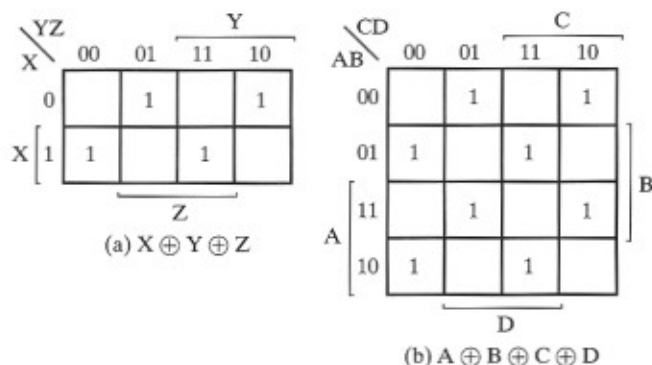
Odd Function

The exclusive-OR operation with three or more variables can be converted into an ordinary Boolean function by replacing the \oplus symbol with its equivalent Boolean expression. In particular, the three-variable case can be converted to a Boolean expression as follows:

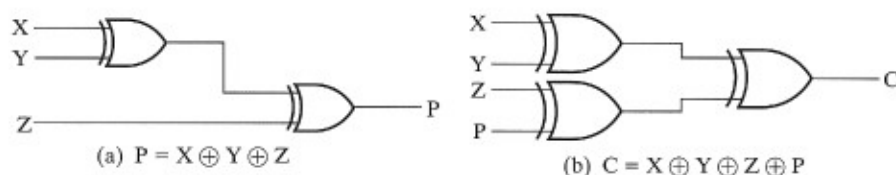
$$\begin{aligned} X \oplus Y \oplus Z &= (X\bar{Y} + \bar{X}Y)\bar{Z} + (XY + \bar{X}\bar{Y})Z \\ &= X\bar{Y}\bar{Z} + \bar{X}Y\bar{Z} + \bar{X}\bar{Y}Z + XYZ \end{aligned}$$

The Boolean expression clearly indicates that the three-variable exclusive-OR is equal to 1 if only one variable is equal to 1 or if all three variables are equal to 1. Hence, whereas in the two-variable function only one variable need be equal to 1, with three or more variables an odd number of variables must be equal to 1. As a consequence, the multiple-variable exclusive-OR operation is defined as the *odd function*. In fact, strictly speaking, this is the correct name for the \oplus operation with three or more variables; the name “exclusive-OR” is applicable to the case with only two variables.

The definition of the odd function can be clarified by plotting the function on a map. Figure 2-31(a) shows the map for the three-variable odd function. The four minterms of the function differ from each other in at least two literals and hence cannot be adjacent on the map. These minterms are said to be *distance two* from each other. The odd function is identified from the four minterms whose binary values have an odd number of 1's. The four-variable case is shown in Figure 2-31(b). The eight minterms marked with 1's in the map constitute the odd function. Note the characteristic pattern of the distance between the 1's in the map. It should be mentioned that the minterms not marked with 1's in the map have an even number of 1's and constitute the complement of the odd function, called the *even function*. The odd function is implemented by means of two-input exclusive-OR gates, as



□ **FIGURE 2-31**
Maps for Multiple-Variable Odd Functions



□ **FIGURE 2-32**
Multiple-Input Odd Functions

shown in Figure 2-32. The even function is obtained by replacing the output gate with an exclusive-NOR gate.

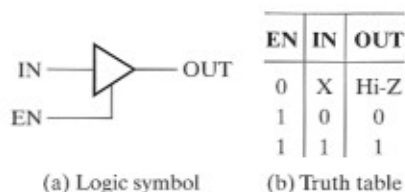
2-9 HIGH-IMPEDANCE OUTPUTS

Thus far, we have considered gates that have only output values logic 0 and logic 1. In this section, we introduce two important structures, three-state buffers and transmission gates, that provide a third output value referred to as the *high-impedance state* and denoted by Hi-Z or just plain Z or z. The Hi-Z value behaves as an open circuit, which means that, looking back into the circuit, we find that the output appears to be disconnected. High impedance outputs may appear on any gate, but here we restrict consideration to two gate structures with single data inputs. Gates with Hi-Z output values can have their outputs connected together, provided that no two gates drive the line at the same time to opposite 0 and 1 values. In contrast, gates with only logic 0 and logic 1 outputs cannot have their outputs connected together.

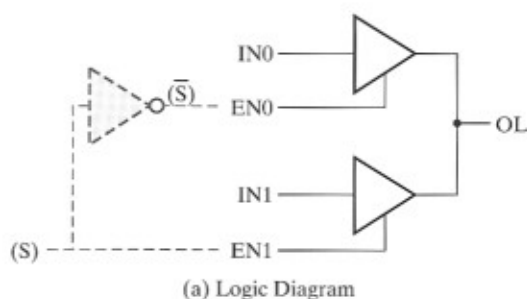
Three-State Buffers The 3-state buffer was introduced earlier as one of the primitive gates. As the name implies, a three-state logic output exhibits three distinct states. Two of the “states” are the logic 1 and logic 0 of conventional logic. The third “state” is the *Hi-Z* value, that, for three-state logic, is referred to as the *Hi-Z state*.

The graphic symbol and truth table for a 3-state buffer are given in Figure 2-33. The symbol in Figure 2-33(a) is distinguished from the symbol for a normal buffer by the enable input, *EN*, entering the bottom of the buffer symbol. From the truth table in Figure 2-33(b), if $EN = 1$, *OUT* is equal to *IN*, behaving like a normal buffer. But for $EN = 0$, the output value is high impedance (*Hi-Z*), regardless of the value of *IN*.

Three-state buffer outputs can be connected together to form a multiplexed output line. Figure 2-34(a) shows two 3-state buffers with their outputs connected to form output line *OL*. We are interested in the output of this structure in terms of the four inputs *EN1*, *EN0*, *IN1*, and *IN0*. The output behavior is given by the truth table in Figure 2-34(b). For *EN1* and *EN0* equal to 0, both buffer outputs are *Hi-Z*. Since both appear as open circuits, *OL* is also an open circuit, represented by a *Hi-Z* value. For $EN1 = 0$ and $EN0 = 1$, the output of the top buffer is *IN0*



□ **FIGURE 2-33**
Three-state Buffer



EN1	EN0	IN1	IN0	OL
0	0	X	X	Hi-Z
(S) 0	(\bar{S}) 1	X	0	0
0	1	X	1	1
1	0	0	X	0
1	0	1	X	1
1	1	0	0	0
1	1	1	1	1
1	1	0	1	
1	1	1	0	

(b) Truth table

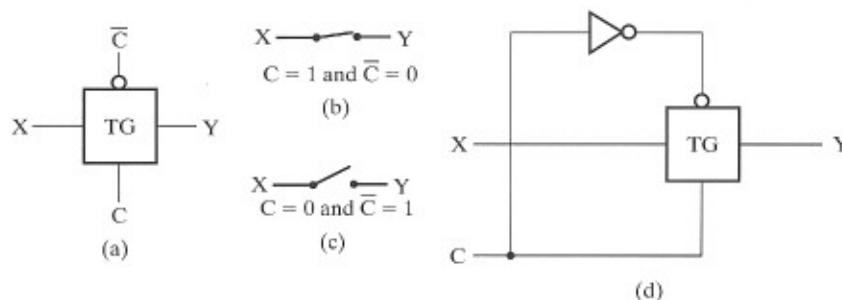
□ **FIGURE 2-34**
Three-state Buffers Forming a Multiplexed Line OL

and the output of bottom buffer is *Hi-Z*. Since the value of *IN0* combined with an open circuit is just *IN0*, *OL* has value *IN0*, giving the second and third rows of the truth table. A corresponding, but opposite, case occurs for *EN1* = 1 and *EN0* = 0, so *OL* has value *IN1*, giving the fourth and fifth rows of the truth table. For *EN1* and *EN0* both 1, the situation is more complicated. If *IN1* = *IN0*, then their mutual value appears at *OL*. But if *IN1* ≠ *IN0*, then their values conflict at the output. The conflict results in an electrical current flowing from the buffer output that is at 1 into the buffer output that is at 0. This current is often large enough to

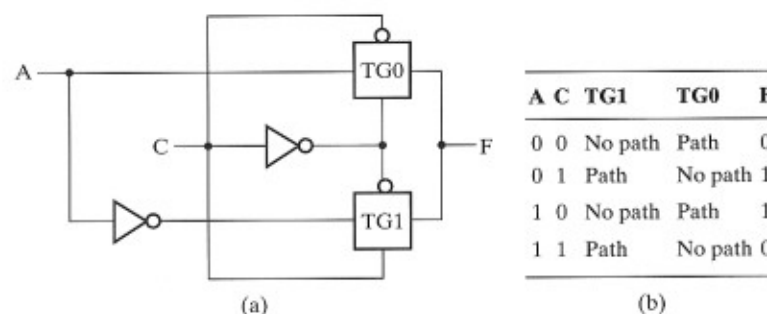
cause heating and may even destroy the circuit, as symbolized by the “smoke” icons in the truth table. Clearly, such a situation must be avoided. The designer must ensure that $EN0$ and $EN1$ never equal 1 at the same time. In the general case, for n 3-state buffers attached to a bus line, EN can equal 1 for only one of the buffers and must be 0 for the rest. One way to ensure this is to use a decoder to generate the EN signals. For the two-buffer case, the decoder is just an inverter with select input S , as shown in dotted lines in Figure 2-34(a). It is interesting to examine the truth table with the inverter in place. It consists of the shaded area of the table in Figure 2-34(b). Clearly, the value on S selects between inputs $IN0$ and $IN1$. Further, the circuit output OL is never in the Hi-Z state.

Transmission Gates In integrated circuit logic, there is a CMOS transistor circuit logic that is important enough to be separately represented at the gate level. This circuit, called a *transmission gate* (TG), is one form of an electronic switch for connecting and disconnecting two points in a circuit. Figure 2-35(a) shows the IEEE symbol for the transmission gate. It has four external connections or ports. C and \bar{C} are the control inputs and X and Y are the signals to be connected or disconnected by the TG. In Figure 2-35(b) and (c), the switch model for the transmission gate appears. If $C = 1$ and $\bar{C} = 0$, X and Y are connected as represented in the model by a “closed” switch and signals can pass from X to Y or from Y to X . If $C = 0$ and $\bar{C} = 1$, X and Y are disconnected as represented in the model by an “open” switch and signals cannot pass between X and Y . In normal use, the control inputs are connected by an inverter as shown in Figure 2-35(d), so that C and \bar{C} are the complements of each other.

To illustrate the use of a transmission gate, an exclusive-OR gate constructed from two transmission gates and two inverters is shown in Figure 2-36(a). Input C controls the paths through the transmission gates, and input A provides the output for F . If input C is equal to 1, a path exists through transmission gate TG1 connecting F to \bar{A} , and no path exists through TG0. If input C is equal to 0, a path exists through TG0 connecting F to A , and no path exists through TG1. Thus, the output F is connected to A . This results in the exclusive-OR truth table, as indicated in Figure 2-36(b).



□ FIGURE 2-35
Transmission Gate (TG)



□ **FIGURE 2-36**
Transmission Gate Exclusive OR

2-10 CHAPTER SUMMARY

The primitive logic operations AND, OR, and NOT define three primitive logic components called gates, from which digital systems are implemented. A Boolean algebra defined in terms of these operations provides a tool for manipulating Boolean functions in designing digital logic circuits. Minterm and maxterm standard forms correspond directly to truth tables for functions. These standard forms can be manipulated into sum-of-products and product-of-sums forms, which correspond to two-level gate circuits. Two cost measures to be minimized in optimizing a circuit are the number of input literals to the circuit and the total number of inputs to the gates in the circuit. K-maps with two to four variables are an effective alternative to algebraic manipulation in optimizing small circuits. These maps can be used to optimize sum-of-products forms, product-of-sums forms, and incompletely specified functions with don't-care conditions. Transforms for optimizing multiple level circuits with three or more levels of gating are illustrated.

The primitive operations AND and OR are not directly implemented by primitive logic elements in the most popular logic family. Thus, NAND and NOR primitives as well as complex gates that implement these families are introduced. A more complex primitive, the exclusive-OR, as well as its complement, the exclusive-NOR, are presented along with their mathematical properties.

REFERENCES

1. BOOLE, G. *An Investigation of the Laws of Thought*. New York: Dover, 1854.
2. KARNAUGH, M. "A Map Method for Synthesis of Combinational Logic Circuits," *Transactions of AIEE, Communication and Electronics*, 72, part I (Nov. 1953), 593-99.
3. DIETMEYER, D. L. *Logic Design of Digital Systems*, 3rd ed. Boston: Allyn Bacon, 1988.

4. MANO, M. M. *Digital Design*, 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2002.
5. ROTH, C. H. *Fundamentals of Logic Design*, 4th ed. St. Paul: West, 1992.
6. HAYES, J. P. *Introduction to Digital Logic Design*. Reading, MA: Addison-Wesley, 1993.
7. WAKERLY, J. F. *Digital Design: Principles and Practices*, 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2000.
8. GAJSKI, D. D. *Principles of Digital Design*. Upper Saddle River, NJ: Prentice Hall, 1997.
9. *IEEE Standard Graphic Symbols for Logic Functions*. (Includes IEEE Std 91a-1991 Supplement and IEEE Std 91-1984.) New York: The Institute of Electrical and Electronics Engineers, 1991.

PROBLEMS



The plus (+) indicates a more advanced problem and the asterisk (*) indicates a solution is available on the Companion Website for the text.

- 2-1. *Demonstrate by means of truth tables the validity of the following identities:
 - (a) DeMorgan's theorem for three variables: $\overline{XYZ} = \overline{X} + \overline{Y} + \overline{Z}$
 - (b) The second distributive law: $X + YZ = (X + Y)(X + Z)$
 - (c) $\overline{XY} + \overline{YZ} + \overline{XZ} = \overline{X}\overline{Y} + \overline{Y}\overline{Z} + \overline{X}\overline{Z}$
- 2-2. *Prove the identity of each of the following Boolean equations, using algebraic manipulation:
 - (a) $\overline{X}\overline{Y} + \overline{X}Y + XY = \overline{X} + Y$
 - (b) $\overline{A}B + \overline{B}C + AB + \overline{B}C = 1$
 - (c) $Y + \overline{X}Z + X\overline{Y} = X + Y + Z$
 - (d) $\overline{X}\overline{Y} + \overline{Y}Z + XZ + XY + Y\overline{Z} = \overline{X}\overline{Y} + XZ + Y\overline{Z}$
- 2-3. +Prove the identity of each of the following Boolean equations, using algebraic manipulation:
 - (a) $AB + B\overline{C}\overline{D} + \overline{A}BC + \overline{C}D = B + \overline{C}D$
 - (b) $WY + \overline{W}Y\overline{Z} + WXZ + \overline{W}X\overline{Y} = WY + \overline{W}X\overline{Z} + \overline{X}Y\overline{Z} + X\overline{Y}Z$
 - (c) $A\overline{C} + \overline{A}B + \overline{B}C + \overline{D} = (\overline{A} + \overline{B} + \overline{C} + \overline{D})(A + B + C + \overline{D})$
- 2-4. +Given that $A \cdot B = 0$ and $A + B = 1$, use algebraic manipulation to prove that

$$(A + C) \cdot (\overline{A} + B) \cdot (B + C) = B \cdot C$$

- 2-5. +A specific Boolean algebra with just two elements 0 and 1 has been used in this chapter. Other Boolean algebras can be defined with more than two elements by using elements that correspond to binary strings. These algebras form the mathematical foundation for bitwise logical operations

that we will study in Chapter 7. Suppose that the strings are each a nibble (half of a byte) of four bits. Then there are 2^4 , or 16, elements in the algebra, where an element I is the 4-bit nibble in binary corresponding to I in decimal. Based on bitwise application of the two-element Boolean algebra, define each of the following for the new algebra so that the Boolean identities hold:

- (a) The OR operation $A + B$ for any two elements A and B
 - (b) The AND operation $A \cdot B$ for any two elements A and B
 - (c) The element that acts as the 0 for the algebra
 - (d) The element that acts as the 1 for the algebra
 - (e) For any element A , the element \bar{A} .
- 2-6. Simplify the following Boolean expressions to expressions containing a minimum number of literals:
- (a) $\bar{A}\bar{C} + \bar{A}BC + \bar{B}C$
 - (b) $(A + B)(\bar{A} + \bar{B})$
 - (c) $ABC + \bar{A}C$
 - (d) $BC + B(AD + \bar{C}D)$
 - (e) $(B + \bar{C} + B\bar{C})(BC + A\bar{B} + AC)$
- 2-7. *Reduce the following Boolean expressions to the indicated number of literals:
- (a) $\bar{X}\bar{Y} + XYZ + \bar{X}Y$ to three literals
 - (b) $X + Y(Z + \bar{X} + \bar{Z})$ to two literals
 - (c) $\bar{W}X(\bar{Z} + \bar{Y}Z) + X(W + \bar{W}YZ)$ to one literal
 - (d) $(AB + \bar{A}\bar{B})(\bar{C}\bar{D} + CD) + \bar{A}\bar{C}$ to four literals
- 2-8. Using DeMorgan's theorem, express the function
- $$F = \bar{A}BC + \bar{B}\bar{C} + A\bar{B}$$
- (a) with only OR and complement operations.
 - (b) with only AND and complement operations.
- 2-9. *Find the complement of the following expressions:
- (a) $A\bar{B} + \bar{A}B$
 - (b) $(\bar{V}W + X)Y + \bar{Z}$
 - (c) $WX(\bar{Y}Z + Y\bar{Z}) + \bar{W}\bar{X}(\bar{Y} + Z)(Y + \bar{Z})$
 - (d) $(A + \bar{B} + C)(\bar{A}\bar{B} + C)(A + \bar{B}\bar{C})$
- 2-10. *Obtain the truth table of the following functions, and express each function in sum-of-minterms and product-of-maxterms form:
- (a) $(XY + Z)(Y + XZ)$
 - (b) $(\bar{A} + B)(\bar{B} + C)$
 - (c) $WX\bar{Y} + WX\bar{Z} + WXZ + Y\bar{Z}$

2-11. For the Boolean functions E and F , as given in the following truth table:

X	Y	Z	E	F
0	0	0	1	0
0	0	1	0	0
0	1	0	1	1
0	1	1	0	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	1
1	1	1	0	1

- (a) List the minterms and maxterms of each function.
 - (b) List the minterms of \bar{E} and \bar{F} .
 - (c) List the minterms of $E + F$ and $E \cdot F$.
 - (d) Express E and F in sum-of-minterms algebraic form.
 - (e) Simplify E and F to expressions with a minimum of literals.
- 2-12.** *Convert the following expressions into sum-of-products and product-of-sums forms:
- (a) $(AB + C)(B + \bar{C}D)$
 - (b) $\bar{X} + X(X + \bar{Y})(Y + \bar{Z})$
 - (c) $(A + B\bar{C} + CD)(\bar{B} + EF)$
- 2-13.** Draw the logic diagram for the following Boolean expressions. The diagram should correspond exactly to the equation. Assume that the complements of the inputs are not available.
- (a) $W\bar{X}\bar{Y} + \bar{W}Z + YZ$
 - (b) $A(B\bar{D} + \bar{B}D) + D(BC + \bar{B}\bar{C})$
 - (c) $W\bar{Y}(X + Z) + \bar{X}Z(W + Y) + W\bar{X}(Y + Z)$
- 2-14.** Optimize the following Boolean functions by means of a three-variable map:
- (a) $F(X, Y, Z) = \sum m(1, 3, 6, 7)$
 - (b) $F(X, Y, Z) = \sum m(3, 5, 6, 7)$
 - (c) $F(A, B, C) = \sum m(0, 1, 2, 4, 6)$
 - (d) $F(A, B, C) = \sum m(0, 3, 4, 5, 7)$
- 2-15.** *Optimize the following Boolean expressions using a map:
- (a) $\bar{X}\bar{Z} + Y\bar{Z} + XYZ$
 - (b) $\bar{A}B + \bar{B}C + \bar{A}\bar{B}\bar{C}$
 - (c) $\bar{A}\bar{B} + A\bar{C} + \bar{B}C + \bar{A}B\bar{C}$

2-16. Optimize the following Boolean functions by means of a four-variable map:

(a) $F(A, B, C, D) = \Sigma m(2, 3, 8, 9, 10, 12, 13, 14)$

(b) $F(W, X, Y, Z) = \Sigma m(0, 2, 5, 6, 8, 10, 13, 14, 15)$

(c) $F(A, B, C, D) = \Sigma m(0, 2, 3, 7, 8, 10, 12, 13)$

2-17. Optimize the following Boolean functions, using a map:

(a) $F(W, X, Y, Z) = \Sigma m(0, 2, 5, 8, 9, 11, 12, 13)$

(b) $F(A, B, C, D) = \Sigma m(3, 4, 6, 7, 9, 12, 13, 14, 15)$

2-18. *Find the minterms of the following expressions by first plotting each expression on a map:

(a) $XY + XZ + \bar{X}YZ$

(b) $XZ + \bar{W}X\bar{Y} + WXY + \bar{W}YZ + W\bar{Y}Z$

(c) $\bar{B}\bar{D} + ABD + \bar{A}BC$

2-19. *Find all the prime implicants for the following Boolean functions, and determine which are essential:

(a) $F(W, X, Y, Z) = \Sigma m(0, 2, 5, 7, 8, 10, 12, 13, 14, 15)$

(b) $F(A, B, C, D) = \Sigma m(0, 2, 3, 5, 7, 8, 10, 11, 14, 15)$

(c) $F(A, B, C, D) = \Sigma m(1, 3, 4, 5, 9, 10, 11, 12, 13, 14, 15)$

2-20. Optimize the following Boolean functions by finding all prime implicants and essential prime implicants and applying the selection rule:

(a) $F(W, X, Y, Z) = \Sigma m(0, 1, 4, 5, 7, 8, 9, 12, 14, 15)$

(b) $F(A, B, C, D) = \Sigma m(1, 5, 6, 7, 11, 12, 13, 15)$

(c) $F(W, X, Y, Z) = \Sigma m(0, 2, 3, 4, 5, 7, 8, 10, 11, 12, 13, 15)$

2-21. Optimize the following Boolean functions in product-of-sums form:

(a) $F(W, X, Y, Z) = \Sigma m(0, 2, 3, 4, 8, 10, 11, 15)$

(b) $F(A, B, C, D) = \Pi M(0, 2, 4, 5, 8, 10, 11, 12, 13, 14)$

2-22. *Optimize the following expressions in (1) sum-of-products and (2) product-of-sums forms:

(a) $A\bar{C} + \bar{B}D + \bar{A}CD + ABCD$

(b) $(\bar{A} + \bar{B} + \bar{D})(A + \bar{B} + \bar{C})(\bar{A} + B + \bar{D})(B + \bar{C} + \bar{D})$

(c) $(\bar{A} + \bar{B} + D)(\bar{A} + \bar{D})(A + B + \bar{D})(A + \bar{B} + C + D)$

2-23. Optimize the following functions into (1) sum-of-products and (2) product-of-sums forms:

(a) $F(A, B, C, D) = \Sigma m(2, 3, 5, 7, 8, 10, 12, 13)$

(b) $F(W, X, Y, Z) = \Pi M(2, 10, 13)$

- 2-24.** Optimize the following Boolean functions F together with the don't-care conditions d :
- (a) $F(A, B, C, D) = \Sigma m(0, 3, 5, 7, 11, 13)$, $d(A, B, C, D) = \Sigma m(4, 6, 14, 15)$
 - (b) $F(W, X, Y, Z) = \Sigma m(0, 6, 8, 13, 14)$, $d(W, X, Y, Z) = \Sigma m(2, 4, 7, 10, 12)$
 - (c) $F(A, B, C) = \Sigma m(0, 1, 2, 4, 5)$, $d(A, B, C) = \Sigma m(3, 6, 7)$
- 2-25.** *Optimize the following Boolean functions F together with the don't-care conditions d . Find all prime implicants and essential prime implicants, and apply the selection rule.
- (a) $F(A, B, C) = \Sigma m(3, 5, 6)$, $d(A, B, C) = \Sigma m(0, 7)$
 - (b) $F(W, X, Y, Z) = \Sigma m(0, 2, 4, 5, 8, 14, 15)$, $d(W, X, Y, Z) = \Sigma m(7, 10, 13)$
 - (c) $F(A, B, C, D) = \Sigma m(4, 6, 7, 8, 12, 15)$,
 $d(A, B, C, D) = \Sigma m(2, 3, 5, 10, 11, 14)$
- 2-26.** Optimize the following Boolean functions F together with the don't-care conditions d in (1) sum-of-products and (2) product-of-sums form:
- (a) $F(A, B, C, D) = \Pi M(1, 3, 4, 6, 9, 11)$,
 $d(A, B, C, D) = \Sigma m(0, 2, 5, 10, 12, 14)$
 - (b) $F(W, X, Y, Z) = \Sigma m(3, 4, 9, 15)$, $d(W, X, Y, Z) = \Sigma m(0, 2, 5, 10, 12, 14)$
- 2-27.** Use decomposition to find minimum gate input count, multiple-level implementations for the functions using AND and OR gates and inverters.
- (a) $F(A, B, C, D) = A\bar{B}C + \bar{A}BC + A\bar{B}D + \bar{A}BD$
 - (b) $F(W, X, Y, Z) = WY + XY + \bar{W}XZ + W\bar{X}Z$
- 2-28.** Use extraction to find a shared, minimum gate input count, multiple-level implementation for the pair of functions given using AND and OR gates and inverters.
- (a) $F(A, B, C, D) = \Sigma m(0, 5, 11, 14, 15)$, $d(A, B, C, D) = \Sigma m(10)$
 - (b) $G(A, B, C, D) = \Sigma m(2, 7, 10, 11, 14)$, $d(A, B, C, D) = \Sigma m(15)$
- 2-29.** Use elimination to flatten each of the function sets given into a two-level sum-of-products form.
- (a) $F(A, B, G, H) = AB\bar{G} + \bar{B}G + \bar{A}\bar{H}$, $G(C, D) = C\bar{D} + \bar{C}D$,
 $H(B, C, D) = B + CD$
 - (b) $T(U, V, Y, Z) = YZU + \bar{Y}\bar{Z}V$, $U(W, X) = W + \bar{X}$,
 $V(W, X, Y) = \bar{W}Y + X$
- 2-30.** *Prove that the dual of the exclusive-OR is also its complement.
- 2-31.** Implement the following Boolean function with exclusive-OR and AND gates, using a minimum number of gate inputs:

$$F(A, B, C, D) = AB\bar{C}D + A\bar{D} + \bar{A}D$$

- 2-32. (a)** Implement function $H = \overline{X}Y + XZ$ using two three-state buffers and an inverter.
- (b)** Construct an exclusive-OR gate by interconnecting two three-state buffers and two inverters.

- 2-33. (a)** Connect the outputs of three 3-state buffers together, and add additional logic to implement the function

$$F = \overline{A} B C + A B D + A \overline{B} \overline{D}$$

Assume that C, D, and \overline{D} are data inputs to the buffers and A and B pass through logic that generates the enable inputs.

- (b)** Is your design in part (a) free of three-state output conflicts? If not, change the design if necessary to be free of such conflicts.
- 2-34.** Use only transmission gates and inverters to implement the function in problem 2-32.
- 2-35.** Depending on the design and the overall logic family being used, it is usually not a good idea to leave the output of a three-state or transmission gate circuit in the high impedance (Hi-Z) state.
- (a)** For the transmission gate circuit designed in problem 2-33, give all input combinations for which the output F is the high-impedance state.
- (b)** Modify the enable logic driving the enable inputs so that the output is either a 0 or a 1 (instead of Hi-Z).

CHAPTER

3

COMBINATIONAL LOGIC DESIGN

In this chapter, we learn about the design of combinational circuits. We introduce the use of a hierarchy and top-down design, both of which are essential to the design of digital circuits. Further, computer-aided design is briefly discussed, including hardware description languages (HDLs) and logic synthesis, two concepts with crucial roles in the efficient design of modern, complex circuits.

Concepts related to the underlying technology for digital circuit implementation are covered in the design space section. The properties of logic gates, including integration levels, logic families, and parameters for logic technologies, are presented. Fan-in, fan-out, and propagation delay for gates are defined, and the positive and negative logic concepts are introduced. Finally, trade-offs between dimensions of the design space, such as cost and performance, are touched upon.

A design procedure with five major steps is presented. The first three steps, specification, formulation, and optimization are illustrated by examples. Fixed implementation technologies are introduced, and technology mapping for these technologies, the next step of the design procedure, is defined and illustrated. The final step of the design procedure, verification, is illustrated by an example using both a manual method and logic simulation. The chapter concludes with an introduction to programmable implementation technologies.

The various concepts in this chapter are pervasive in the design of the generic computer in the diagram at the beginning of Chapter 1. Concepts from this chapter apply across all of the digital components of the generic computer including memories.

3-1 DESIGN CONCEPTS AND AUTOMATION

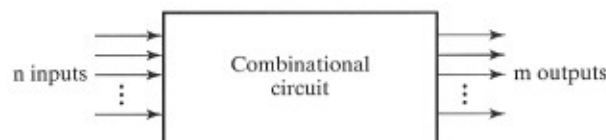
In Chapter 1, we learned about binary numbers and binary codes that represent discrete quantities of information. In Chapter 2, we introduced the various logic

gates and learned how to optimize Boolean expressions and equations in order to achieve economical gate implementations. The purpose of this chapter is to use the knowledge acquired in the previous chapters to formulate systematic design procedures for combinational circuits. In addition, the design steps will be related to the use of computer-aided design tools. The various examples introduced provide design practice. Modern digital design involves a number of techniques and tools that are essential to the design of complex circuits and systems. Design hierarchy, top-down design, computer-aided design tools, hardware description languages, and logic synthesis, are among the most important tools for effective and efficient digital design.

Logic circuits for digital systems may be combinational or sequential. A combinational circuit consists of logic gates whose outputs at any time are determined by logic operations on the input values. A combinational circuit performs an operation that can be specified logically by a set of Boolean equations. In contrast, sequential circuits employ elements that store bit values. Sequential circuit outputs are functions of the inputs and the bit values in the storage elements, which, in turn, are a function of previously applied inputs and stored values. As a consequence, the outputs of a sequential circuit depend not only on the current input values, but also on past inputs. The behavior of the circuit must be specified by a sequence in time of inputs and internal stored bit values. Sequential circuits are presented in Chapter 6.

A combinational circuit consists of input variables, output variables, logic gates, and interconnections. The interconnected logic gates accept signals from the inputs and generate signals at the outputs. A block diagram of a combinational circuit is shown in Figure 3-1. The n input variables come from the environment of the circuit, and the m output variables are available for use by the environment. Each input and output variable exists physically as a binary signal that takes on values logic 1 or logic 0.

For n input variables, there are 2^n possible binary input combinations. For each binary combination of the input variables, there is one possible binary value on each output. Thus, a combinational circuit can be specified by a truth table that lists the output values for each combination of the input variables. A combinational circuit can also be described by m Boolean functions, one for each output variable. Each such function is expressed as a function of the n input variables. Before defining the design process, we introduce two fundamental concepts related to design: design hierarchy and top-down design.



□ **FIGURE 3-1**
Block Diagram of a Combinational Circuit

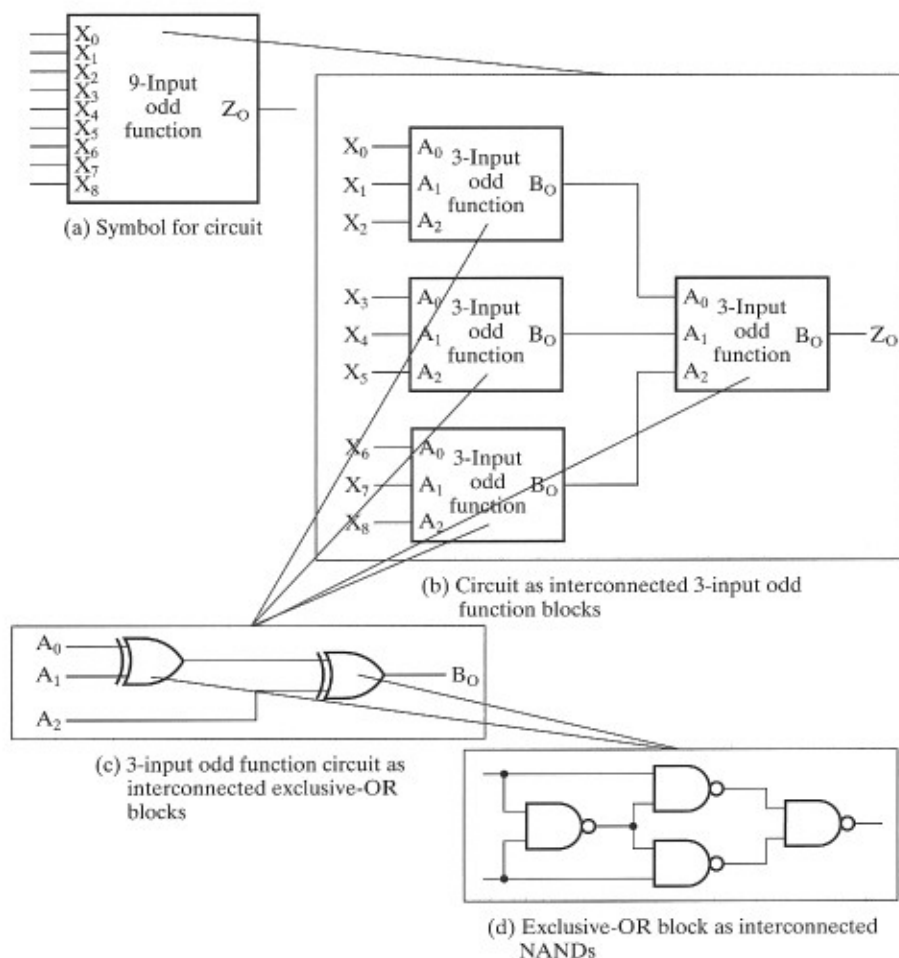
Design Hierarchy

A circuit may be specified by a symbol showing its inputs and outputs and a description defining exactly how it operates. In terms of implementation, however, a circuit is composed of logic gates or other logic structures that are interconnected. A complex digital system may contain millions of interconnected gates. In fact, a single very large-scale integrated (VLSI) processor circuit often contains several tens of millions of gates. With such complexity, the interconnected gates appear to be an incomprehensible maze. Thus, no complex system or circuit can be designed simply by interconnecting gates one at a time. In order to deal with such circuit complexity, a “divide and conquer” approach is used. The circuit is broken up into pieces we call *blocks*. The blocks are interconnected to form the circuit. The functions of these blocks and their interfaces are carefully defined, so that the circuit formed by interconnecting the blocks obeys the circuit specification. If a block is still too large and complex to be designed as a single entity, it can be broken into smaller blocks. This process can be repeated as necessary. Note that since we are working primarily with logic circuits, we use the term “circuit” in this discussion, but the ideas apply equally well to the “systems” covered in later chapters.

The “divide and conquer” approach is illustrated by a combinational circuit in Figure 3-2. The circuit implements a 9-input odd function. A symbol for this overall circuit is shown in part (a) of the figure. In part (b), a logic diagram, or *schematic*, is given for the circuit represented by the symbol in part (a). In this schematic, the designer has chosen to break up the circuit into four identical blocks, each of which is a 3-input odd function. The symbol for the 3-input odd function is repeated four times. The four symbols are interconnected to form the 9-input odd-function circuit. In part (c), a 3-input odd function block is shown to consist of two interconnected exclusive-OR gates. Finally, in part (d), the exclusive-OR is implemented using NAND gates. Note that in each case, as we move downward from the top level, symbols are replaced by schematics that represent the implementation of the symbol.

This design approach is referred to as *hierarchical design*, and the resulting related symbols and schematics constitute a *hierarchy* representing the circuit designed. The structure of the hierarchy can be represented without the interconnections by starting with the top block and connecting below each block those blocks from which it is made. Using this representation, the hierarchy for the 9-input odd-function circuit is shown in Figure 3-3(a) using this representation. Note that the resulting structure has the form of a tree with the root at the top. The “leaves” of the tree are the NAND gates, in this case 32 of them. In order to provide a more compact representation of the hierarchy, we can reuse blocks as shown in Figure 3-3(b). This diagram corresponds to blocks used in Figure 3-2 with only one copy of each distinct block shown. These diagrams and the circuits in Figure 3-2 are helpful in illustrating a number of useful concepts associated with hierarchies and hierarchical blocks.

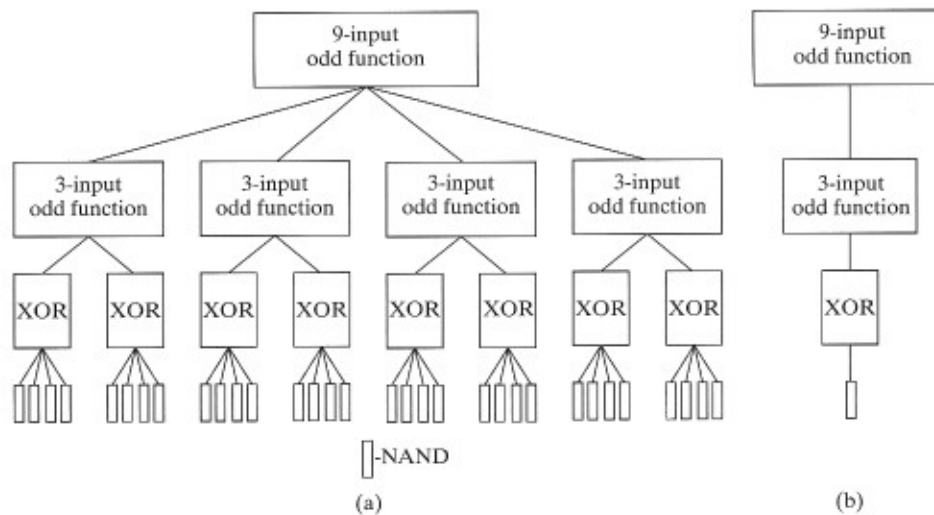
First of all, a hierarchy reduces the complexity required to represent the schematic diagram of a circuit. For example, in Figure 3-3(a), 32 NAND blocks



□ **FIGURE 3-2**
Example of Design Hierarchy and Reusable Blocks

appear. This means that if a 9-input odd-function circuit was designed directly in terms of NAND gates, the schematic for the circuit would consist of 32 interconnected NAND gate symbols, in contrast to just 10 symbols used to describe the circuit implementation as a hierarchy in Figure 3-2. Thus, a hierarchy gives a simplified representation of a complex circuit.

Second, the hierarchy ends at a set of “leaves” in Figure 3-3. In this case, the leaves consist of NAND gates. Since the NAND gates are electronic circuits, and we are interested here only in designing the logic, the NAND gates are commonly called *primitive blocks*. These are rudimentary blocks, such as gates, that have a symbol, but no logic schematic. Primitive blocks are a rudimentary type of *pre-defined blocks*. In general, more complex structures that likewise have symbols,



□ **FIGURE 3-3**
Diagrams Representing the Hierarchy for Figure 3-2

but no logic schematics, are also predefined blocks. Instead of schematics, their function can be defined by a program or description that can serve as a model. For example, in the hierarchy depicted in Figure 3-3, the exclusive-OR gates could have been considered as predefined blocks. In such a case, the diagram describing the exclusive-OR blocks in Figure 3-2(d) would not be necessary. The hierarchical representations in Figure 3-3 would then end with the exclusive-OR blocks. In any hierarchy, the “leaves” consist of predefined blocks, some of which may be primitives.

A third very important property that results from hierarchical design is the reuse of blocks as illustrated in Figures 3-3(a) and (b). In part (a), there are four copies of the 3-input odd-function block and eight copies of the exclusive-OR block. In part (b), there is only one copy of the 3-input odd-function block and one copy of the exclusive-OR block. This represents the fact that the designer has to design only one 3-input odd-function block and one exclusive-OR block and can use these blocks four times and eight times, respectively, in the 9-input odd-function circuit. In general, suppose that at various levels of the hierarchy, the blocks used are carefully defined in such a manner that many of them are identical. For these repeated blocks, only one design is necessary. This design can be used everywhere the block is required. The appearance of a block within a design is called an *instance* of the block and its use is called an *instantiation*. The block is *reusable* in the sense that it can be used in multiple places in the circuit design and, possibly, in the design of other circuits as well. This concept greatly reduces the design effort required for complex circuits. Note that, in the implementation of the circuit, separate hardware has to be provided for each instance of the block as represented in Figure 3-3(a). The reuse, as represented in Figure 3-3(b), is confined to the schematics, not to the actual hardware implementation.

After completing a discussion of the design process, in Chapters 4 and 5, we focus on predefined, reusable blocks that typically lie at the lower levels of logic design hierarchies. These are blocks that provide basic functions used in digital design. They allow designers to do much of the design process above the primitive block level. We refer to these particular blocks as *functional blocks*. Thus, a functional block is a predefined collection of interconnected gates. Many of these functional blocks have been available for decades as medium-scale integrated (MSI) circuits that were interconnected to form larger circuits or systems. Similar blocks are now in computer-aided design tool libraries used for designing larger integrated circuits. These functional blocks provide a catalog of elementary digital components that are widely used in the design and implementation of integrated circuits for computers and digital systems.

Top-Down Design

Ideally, the design process is performed *top down*. This means that the circuit function is specified typically by text or a hardware description language (HDL), plus constraints on cost, performance, and reliability. At high levels of the design, the circuit is then repeatedly divided into blocks as necessary until the blocks are small enough to perform logic design. For manual logic design, the blocks may need to be further divided. In automated synthesis, the HDL description is converted to logic automatically. Then, for both manual design and automated synthesis, the logic is optimized and then mapped to the available primitive elements. In fact, reality departs significantly from this ideal view, particularly at the higher levels of the design. In order to obtain reusability and to make maximum use of predefined modules, it is often necessary to perform portions of the design *bottom up*. In addition, a particular circuit design obtained during the design process may violate one of the constraints in the initial specification. In this case, it is necessary to backtrack upward through the hierarchy until a level is reached at which the violation can be eliminated. A portion of the design is then revised at that level and the revisions are carried back downward through the hierarchy.

In this text, since reader familiarity with logic and computer design is probably limited, we need to build a ready set of functional blocks to provide direction in top-down design. Likewise, a sense of how to break up a circuit into blocks that can serve to guide the top-down approach also must be mastered. So the focus in much of the text will be on bottom-up rather than top-down design. To begin building the basis for top-down design, in Chapters 4 and 5, we focus our efforts on the design of frequently used functional blocks. In Chapters 7 and 10, we illustrate how larger circuits and systems are broken down into blocks and how these blocks are implemented with functional blocks. Finally, beginning with Chapter 11, we apply these ideas to look at design from more of a top-down perspective.

Computer-Aided Design

Designing complex systems and integrated circuits would not be feasible without the use of *computer-aided design (CAD)* tools. *Schematic capture* tools support the drawing

of blocks and interconnections at all levels of the hierarchy. At the level of primitives and functional blocks, *libraries* of graphics symbols are provided. Schematic capture tools support the construction of a hierarchy by permitting the generation of symbols for hierarchical blocks and the replication of symbols for reuse.

The primitive blocks and the functional block symbols from libraries have associated models that allow the behavior and the timing of the hierarchical blocks and the entire circuit to be verified. This verification is performed by applying inputs to the blocks or circuit and using a *logic simulator* to determine the outputs. We will be illustrating logic simulation in a number of examples.

The primitive blocks from libraries can also have associated data, such as physical area information and delay parameters, that can be used by *logic synthesizers* to optimize designs being generated automatically from hardware description language specifications.

Hardware Description Languages

Thus far, we have mentioned hardware description languages only casually. In modern design, however, such languages have become crucial to the design process. Initially, we justify such languages by describing their uses. We will then briefly discuss VHDL and Verilog®, the most popular of these languages. Beginning in Chapter 4, we introduce both of these languages in detail, although, in any given course, we expect that only one of them will be covered.

Hardware description languages resemble programming languages, but are specifically oriented to describing hardware structures and behavior. They differ markedly from typical programming languages in that they represent extensive parallel operation whereas most programming languages represent serial operation. An obvious use for a hardware description language is to provide an alternative to schematics. When a language is used in this fashion, it is referred to as a *structural description* in which the language describes an interconnection of components. Such a structural description, referred to as a *netlist*, can be used as input to logic simulation just as a schematic is used. For this application, models for each of the primitive blocks are required. If an HDL is used, then these models can also be written in the HDL providing a more uniform, portable representation for simulation input.

The power of an HDL becomes more apparent, however, when it is used to represent more than just schematic information. It can represent Boolean equations, truth tables, and complex operations such as arithmetic. Thus, in top-down design, a very high-level description of an entire system can be precisely specified using an HDL. As a part of the design process, this high-level description can then be refined and partitioned into lower-level descriptions. Ultimately, a final description in terms of primitive components and functional blocks can be obtained as the result of the design process. Note that all of these descriptions can be simulated. Since they represent the same system in terms of function, but not necessarily timing, they should respond by giving the same logic values for the same applied inputs. This vital simulation property supports design verification and is one of the principal reasons for the use of HDLs.

A final major reason for the growth of the use of HDLs is logic synthesis. An HDL description of a system can be written at an intermediate level referred to as a register transfer language (RTL) level. A logic synthesis tool with an accompanying library of components can convert such a description into an interconnection of primitive components that implements the circuit. This replacement of the manual logic design process makes the design of complex logic much more efficient.

Currently, there are two HDLs, VHDL and Verilog, that are widely-used, standard hardware design languages. The language standards are defined, approved, and published by the Institute of Electrical and Electronics Engineers (IEEE). All implementations of these languages must obey their respective standard. This standardization gives HDLs another advantage over schematics. HDLs are portable across computer-aided design tools whereas schematic capture tools are typically unique to a particular vendor. In addition to the standard languages, a number of major companies have their own internal languages, often developed long before the standard languages and incorporating features unique to their particular products.

VHDL stands for VHSIC Hardware Description Language. VHDL was developed under contract for the U. S. Department of Defense as a part of the Very-High-Speed Integrated Circuits (VHSIC) program and subsequently became an IEEE standard language. Verilog® was developed by a company, Gateway Design Automation, which was bought by Cadence® Design Systems, Inc. For a while, Verilog was a proprietary language, but eventually became an IEEE standard language. In this text, we present brief introductions to both VHDL and Verilog. These portions of the text are optional and permit your instructor to cover one of the two languages or neither.

Regardless of the HDL, there is a typical procedure used in employing an HDL description as simulation input. These procedure steps are analysis, elaboration, and initialization, followed finally by the simulation. Analysis and elaboration are typically performed by a compiler similar to those for programming languages. *Analysis* checks the description for violations of the syntax and semantic rules for the HDL and produces an intermediate representation of the design. *Elaboration* traverses the design hierarchy represented by the description; in this process, the design hierarchy is flattened to an interconnection of modules that are described only by their behaviors. The end result of the analysis and elaboration performed by the compiler is a simulation model of the original HDL description. This model is then passed to the simulator for execution. *Initialization* sets all of the variables in the simulation model to specified or default values. *Simulation* executes the simulation model in either batch or interactive mode with inputs specified by the user.

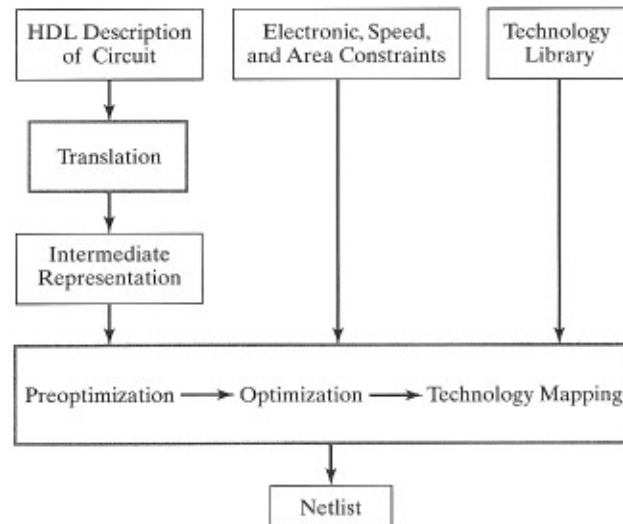
Because of the ability to describe fairly complex hardware efficiently in an HDL, a special HDL structure called a *testbench* may be used. The testbench is a description that includes the design to be tested, typically referred to as the device under test (DUT). The testbench describes a collection of hardware and software functions that apply inputs to the DUT and analyze the outputs for correctness. This approach bypasses the need to provide separate inputs to the simulator and to

analyze, often manually, the simulator outputs. Construction of a testbench provides a uniform verification mechanism that can be used at multiple levels in the top-down design process for verification of correct function of the design.

Logic Synthesis

As indicated earlier, the availability of logic synthesis tools is one of the driving forces behind the growing use of HDLs. Logic synthesis transforms an RTL description of a circuit in an HDL into an optimized netlist representing storage elements and combinational logic. Subsequently, this netlist may be transformed by using physical design tools into an actual integrated circuit layout. This layout serves as the basis for integrated circuit manufacture. The logic synthesis tool takes care of a large portion of the details of a design and allows exploration of the cost/performance trade-offs essential to advanced designs.

Figure 3-4 gives a simple high-level flow of the steps involved in logic synthesis. The user provides an HDL description of the circuit to be designed as well as various constraints or bounds on the design. Electrical constraints include allowable gate fanouts and output loading restrictions. Area and speed constraints direct the optimization steps of the synthesis. Area constraints typically give the maximum permissible area that a circuit is allowed to occupy within the integrated circuit. Alternatively, a general directive may be given which specifies that area is to be minimized. Speed constraints are typically maximum allowable values for propagation delay on various paths in the circuit. Alternatively, a general directive may be given to maximize speed. Area and speed both translate into the cost of a circuit. A



□ **FIGURE 3-4**
High-Level Flow for Logic Synthesis Tool

fast circuit will typically have larger area and thus cost more to manufacture. A circuit that need not operate fast can be optimized for area, and, relatively speaking, costs less to manufacture. In some sophisticated synthesis tools, power consumption can also be used as a constraint. Additional information used by a synthesis tool is a technology library that describes the logic elements available for use in the netlist as well as their delay and loading properties. The latter information is essential in meeting constraints and performing optimization.

The first major step in the synthesis process in Figure 3-4 is a translation of the HDL description into an intermediate form. The information in this representation may be an interconnection of generic gates and storage elements, not taken from a library of primitive blocks, called a *technology library*. It may also be in an alternate form that represents clusters of logic and the interconnections between the clusters.

The second major step in the synthesis process is optimization. A preoptimization step may be used to simplify the intermediate form. For example, logic that is identical in the intermediate form may be shared. Next is the optimization in which the intermediate form is processed to attempt to meet the constraints specified. Typically, two-level and multiple-level optimization are performed. Optimization is followed by *technology mapping* which replaces AND gates, OR gates, and inverters with gates from the technology library. In order to evaluate area and speed parameters associated with these gates, additional information from the technology library is used. In sophisticated synthesis tools, further optimization may be applied during technology mapping in order to improve the likelihood of meeting the constraints on the design. Optimization can be a very complex, time consuming process for large circuits. Many optimization passes may be necessary to achieve the desired results or to demonstrate that constraints are difficult, if not impossible, to meet. The designer may need to modify the constraints or the HDL in order to achieve a satisfactory design. Modification of the HDL may include manual design of some portions of the logic in order to achieve the design goals.

The output of the optimization/technology mapping processes is typically a netlist corresponding to a schematic diagram made up of storage elements, gates, and other combinational logic functional blocks. This output serves as input to physical design tools that physically place the logic elements and route the interconnections between them to produce the layout of the circuit for manufacture. In the case of programmable parts such as field-programmable gate arrays as discussed in Section 3-6, an analog to the physical design tools produces the binary information used to program the logic within the parts.

3-2 THE DESIGN SPACE

For a given design, there is typically a target implementation technology that specifies the primitive elements available and the properties of those elements. In addition, there is a set of constraints that applies to the design. This section deals with the potential primitive gate functions and properties and briefly discusses design constraints and the trade-offs that must be considered while attempting to meet the constraints.

Gate Properties

Digital circuits are constructed with integrated circuits. An integrated circuit (abbreviated IC) is a silicon semiconductor crystal, informally called a chip, containing the electronic components for the digital gates and storage elements. The various components are interconnected on the chip. The chip is mounted in a ceramic or plastic container, and connections are welded from the chip to the external pins to form the integrated circuit. The number of pins may range from 14 on a small IC package to several hundred on a large package. Each IC has a numeric designation printed on the surface of the package for identification. Each vendor publishes datasheets or a catalog that contains the description and all the necessary information about the ICs that it manufactures. Typically, this information is available on vendor websites.

Levels of Integration

As the technology of ICs has improved, the number of gates present in a single silicon chip has increased considerably. Customary reference to a package as being either a small-, medium-, large-, or very large-scale integrated device is used to differentiate between chips with just a few internal gates and those with thousands to tens of millions of gates.

Small-scale integrated (SSI) devices contain several independent primitive gates in a single package. The inputs and outputs of the gates are connected directly to the pins in the package. The number of gates is usually less than 10 and is limited by the number of pins available on the IC.

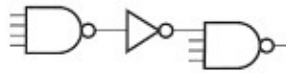
Medium-scale integrated (MSI) devices have approximately 10 to 100 gates in a single package. They usually perform specific elementary digital functions, such as the addition of four bits. MSI digital functions are similar to the functional blocks described in Chapters 4 and Chapter 5.

Large-scale integrated (LSI) devices contain between 100 and a few thousand gates in a single package. They include digital systems such as small processors, small memories, and programmable modules.

Very large-scale integrated (VLSI) devices contain several thousand to tens of millions of gates in a single package. Examples are complex microprocessor and digital signal processing chips. Because of their small transistor dimensions, high density, and comparatively low cost, VLSI devices have revolutionized digital system and computer design. VLSI technology gives designers the capability to create complex structures that previously were not economical to manufacture.

Circuit Technologies

Digital integrated circuits are classified not only by their function, but also by their specific implementation technology. Each technology has its own basic electronic device and circuit structures upon which more complex digital circuits and functions are developed. The specific electronic devices used in the construction of the basic circuits provide the name for the technology. Currently, silicon-based



□ **FIGURE 3-5**
Implementation of a 7-input NAND Gate
using NAND Gates with 4 or Fewer Inputs

Complementary Metal Oxide Semiconductor (CMOS) technology dominates due to its high circuit density, high performance, and low power consumption. Alternative technologies based on Gallium Arsenide (GaAs) and Silicon Germanium (SiGe) are used selectively for very high speed circuits.

Technology Parameters

For each specific implementation technology, there are details that differ in their electronic circuit design and circuit parameters. The most important parameters used to characterize an implementation technology follow:

Fan-in specifies the number of inputs available on a gate.

Fan-out specifies the number of standard loads driven by a gate output. *Maximum Fan-out* for an output specifies the fan-out that the output can drive without impairing gate performance. Standard loads may be defined in a variety of ways depending upon the technology.

Noise margin is the maximum external noise voltage superimposed on a normal input value that will not cause an undesirable change in the circuit output.

Cost for a gate specifies a measure of its contribution to the cost of the integrated circuit containing it.

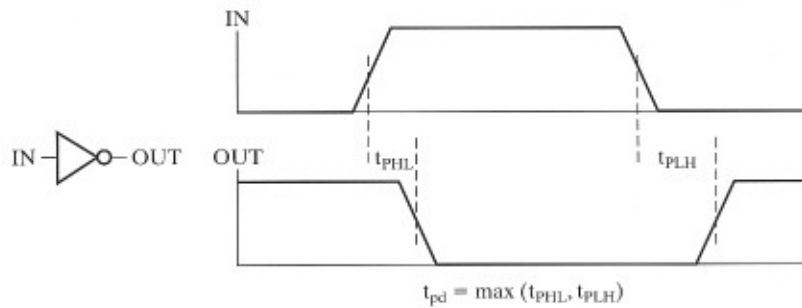
Propagation delay is the time required for a change in value of a signal to propagate from input to output. The operating speed of a circuit is inversely related to the longest propagation delays through the gates of the circuit.

Power dissipation is the power drawn from the power supply and consumed by the gate. The power consumed is dissipated as heat, so the power dissipation must be considered in relation to the operating temperature and cooling requirements of the chip.

Although all of these parameters are important to the designer, further details on only selected parameters are provided next.

FAN-IN For high-speed technologies, *fan-in*, the number of inputs to a gate, is often restricted on gate primitives to no more than four or five. This is primarily due to electronic considerations related to gate speed. To build gates with larger fan-in, interconnected gates with lower fan-in are used during technology mapping. A mapping for a 7-input NAND gate illustrated in Figure 3-5 is made up of two 4-input NANDs and an inverter.

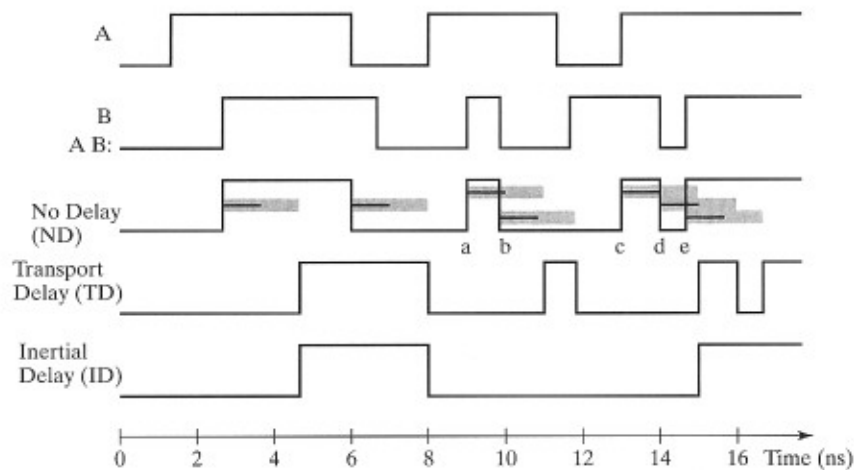
PROPAGATION DELAY The determination of propagation delay is illustrated in Figure 3-6. Three propagation delay parameters are defined. The *high-to-low*



□ **FIGURE 3-6**
Propagation Delay for an Inverter

propagation time t_{PHL} is the delay measured from the reference voltage on the input IN to the reference voltage on the output OUT, with the output voltage going from H to L. The reference voltage we are using is the 50% point, halfway between the minimum and the maximum values of the voltage signals; other reference voltages may be used, depending on the logic family. The *low-to-high propagation time* t_{PLH} is the delay measured from the reference voltage on the input voltage IN to the reference voltage on the output voltage OUT, with the output voltage going from L to H. We define the *propagation delay* t_{pd} as the maximum of these two delays. The reason we have chosen the maximum value is that we will be most concerned with finding the longest time for a signal to propagate from inputs to outputs. Otherwise, the definitions given for t_{pd} may be inconsistent, depending on the use of the data. Manufacturers usually specify the maximum and typical values for both t_{PHL} and t_{PLH} or for t_{pd} for their products.

Two different models, transport delay and inertial delay, are employed in modeling gates during simulation. For *transport delay*, the change in an output in response to the change of an input occurs after a specified propagation delay. *Inertial delay* is similar to transport delay, except that if the input changes cause the output to change twice in an interval less than the *rejection time*, then the first of the two output changes does not occur. The rejection time is a specified value no larger than the propagation delay and is often equal to the propagation delay. An AND gate modeled with both a transport delay and an inertial delay is illustrated in Figure 3-7. To help visualize the delay behavior, we have also given the AND output with no delay. A colored bar on this waveform shows a 2 ns propagation delay time after each input change and a smaller black bar shows a rejection time of 1 ns. The output modeled with the transport delay is identical to that for no delay, except that it is shifted to the right by 2 ns. For the inertial delay, the waveform is likewise shifted. To define the waveform for the delayed output, we will call each change in a waveform an *edge*. To determine whether a particular edge appears in the ID output, it must be determined whether a second edge occurs in the ND output before the end of the rejection time for the edge in question, and whether the edge will result in a change in the ID output. Since edge b occurs before the end of the rejection time for edge a in the ND output, edge a does not



□ **FIGURE 3-7**
Examples of Behavior of Transport and Inertial Delays

appear in the ID output. Since edge b does not change the state of ID, it is ignored. Since edge d occurs at the rejection time after edge c in the ND output, edge c does appear. Edge e, however, occurs within the rejection time after edge d, so edge d does not appear. Since edge c appeared and edge d did not appear, edge e does not cause a change.

FAN-OUT One approach to measuring fan-out is the use of a *standard load*. Each input on a driven gate provides a load on the output of the driving gate which is measured in standard load units. For example, the input to a specific inverter can have load equal to 1.0 standard load. If a gate drives six such inverters, then the fan-out is equal to 6.0 standard loads. In addition, the output of a gate has a maximum load that it can drive called its maximum fan-out. The determination of the maximum fan-out is a function of the particular logic family. Our discussion will be restricted to CMOS, currently the most popular logic family. For CMOS gates, the loading of a gate output by the integrated circuit wiring and the inputs of other gates is modeled as a capacitance. This capacitive loading has no effect on the logic levels as loading often does for other families. Instead, the load on the output of a gate determines the time required for the output of the gate to change from L to H and from H to L. If the load on the output is increased, then this time, called the *transition time*, increases. Thus, the maximum fan-out for a gate is the number of standard loads of capacitance that can be driven with the transition time no greater than its maximum allowable value. For example, a gate with a maximum fan-out of 8 standard loads could drive up to 8 inverters that present 1.0 standard load on each on their inputs.

Because it represents capacitance driven, the actual fan-out of the gate, in terms of standard loads, also affects the propagation delays of the gate. Thus, a

simple expression for propagation delay can be given by a formula or table that considers a fixed delay plus a delay per standard load times the number of standard loads driven, as shown in the next example.

EXAMPLE 3-1 Calculation of Gate Delay Based on Fan-Out

A 4-input NAND gate output is attached to the inputs of the following gates with the given number of standard loads representing their inputs:

4-input NOR gate - 0.80 standard load

3-input NAND gate - 1.00 standard load, and

inverter - 1.00 standard load.

The formula for the delay of the 4-input NAND gate is

$$t_{pd} = 0.07 + 0.021 \times SL \text{ ns}$$

where SL is the sum of the standard loads driven by the gate.

Ignoring the wiring delay, the delay projected for the NAND gate as loaded is

$$t_{pd} = 0.07 + 0.021 \times (0.80 + 1.00 + 1.00) = 0.129 \text{ ns}$$

In modern high-speed circuits, the portion of the gate delay due to wiring capacitance is often significant. While ignoring such delay is unwise, it is difficult to evaluate since it depends on the layout of the wires in the integrated circuit. Nevertheless, since we do not have this information or a method to obtain a good estimate of it, we ignore this delay component here. ■

Both fan-in and fan-out must be dealt with in the technology mapping step of the design process. Gates with fan-ins larger than those available for technology mapping can be implemented with multiple gates. Gates with fan-outs that either exceed their maximum allowable fan-out or have too high a delay need to be replaced with multiple gates or have buffers added at their outputs.

COST For integrated circuits, the cost of a primitive gate is usually based on the area occupied by the layout cell for the circuit. The layout cell area is proportional to the size of the transistors and the wiring in the gate layout. Ignoring the wiring area, the area of the gate is proportional to the number of transistors in the gate, which in turn is usually proportional to the gate input count. If the actual area of the layout is known, then a normalized value of this area provides a more accurate estimation of cost than gate input count.

Positive and Negative Logic

Excluding transitions, the binary signals at the inputs and outputs of any gate have one of two values: H or L. One value represents logic 1 and the other logic 0. There are two different assignments of signal levels to logic values, as shown in Figure 3-8. Choosing the high level H to represent logic 1 defines a *positive-logic* system. Choosing the low level L to represent logic 1 defines a *negative-logic* system. The terms “positive” and “negative” are somewhat misleading, since both signals may be positive voltages or both may be negative voltages. It is not the actual signal values

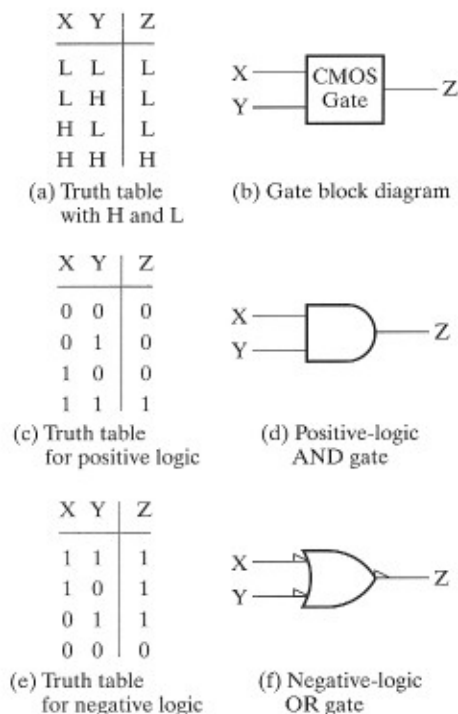
Signal value	Logic value	Signal value	Logic value
H	1	H	0
L	0	L	1

(a) Positive logic (b) Negative logic

□ **FIGURE 3-8**
Signal Assignment and Logic Polarity

that determine the type of logic, but rather the assignment of logic values to the *relative amplitudes* of the two signal ranges.

Integrated circuit data sheets define digital gates in terms of both logic values and signal values H and L. If H and L are used, it is up to the user to decide on a positive or negative logic assignment. Consider, for example, the truth table in Figure 3-9(a). This table is given in a data book for the CMOS gate shown in Figure 3-9(b). The table specifies the physical behavior of the gate when H is 5 volts and L is 0 volts. The truth table of Figure 3-9(c) assumes positive logic, with 1 assigned to H and 0 assigned to L. The table is the same as the truth table for the AND operation. The graphics symbol for a positive-logic AND gate is shown in Figure 3-9(d).



□ **FIGURE 3-9**
Demonstration of Positive and Negative Logic

Now consider the negative logic assignment for the same physical gate, with 1 assigned to L and 0 assigned to H. The result is the truth table of Figure 3-9(e). This table represents the OR operation with the table rows reversed. The graphic symbol for the negative-logic OR gate is shown in Figure 3-9(f). The small triangles on the inputs and output are *polarity indicators*. The presence of a polarity indicator on an input or output signifies that negative logic is assumed for the corresponding signal. Thus, the same physical gate can operate as either a positive-logic AND gate or as a negative-logic OR gate.

The conversion from positive logic to negative logic and vice versa is an operation that changes 1's to 0's and 0's to 1's at both the inputs and the output of a gate. Since this interchange of 1's and 0's is a part of taking the dual, the conversion operation produces the dual of the gate function. Thus, the change of all gate inputs and outputs from one polarity to the other results in taking the dual of the function, with all AND operations (or graphics symbols) converted to OR operations (or graphics symbols) and vice versa. In addition, one must not forget to include the polarity indicator in the graphics symbols when negative logic is assumed, and one must also recognize that the polarity definitions for circuit inputs and circuit outputs have been changed. In this book we do not use negative logic, but assume that all gates operate with a positive-logic assignment.

Design Trade-Offs

Previously, it has been shown that there is a relationship between the actual fan-out of a gate and its propagation delay. Higher fan-out increases the propagation delay. For example, a circuit has a gate G having a fan-out equal to 16.00 standard loads. The delay through this gate, including the delay component based on the standard loads driven, is 0.406 ns. To reduce this delay, a buffer is added to the output of gate G and the 16.00 standard loads are connected to the buffer output. The output of gate G now drives just the buffer. The delay for this combination of the gate and the buffer in series is only 0.323 ns, giving a delay reduction of over 20%. Gate G alone has a cost of 2.0 while gate G plus the buffer have a cost of 3.0. These two circuits illustrate a cost/performance trade-off, the most common of the trade-offs with which a designer must deal. While this example uses two simple circuits, cost/performance trade-offs can be made at much higher levels in a system design. These trade-offs may influence functional specifications for the systems and the implementation approach used for system functions.

Continuing with the simple example, the designer has two choices. If gate G alone is fast enough, it should be selected because of its lower cost. On the other hand, if it is not fast enough, then gate G plus the buffer should be selected. In order to make this trade-off, we need to have one or more constraints on the design of the circuit. Suppose that the constraint is that the maximum input-to-output delay $t_{pd,max} = 0.35$ ns. Since gate G alone does not meet this constraint, the buffer must be added. Alternatively, suppose that the constraint is that the maximum number of area units for the circuit is 2.5. Since gate G plus the buffer do not meet this constraint, gate G alone must be chosen. Finally, suppose that both the delay and area constraints given are expected to be met. Then neither solution is

satisfactory. A new design must be found that meets the constraints or the constraints must be relaxed so that one of the two circuits meets them.

A sample list of possible circuit constraints is as follows:

Maximum input-to-output delay

Maximum area units

Maximum power dissipation

Maximum standard loads presented to circuits driving the circuit inputs

Minimum standard load and drive provided by the circuit outputs

Typically, not all constraints on this list are specified for a given circuit. For example, instead of having both delay and cost constraints, the delay constraint may be fixed with the cost to be minimized while meeting all other constraints.

3-3 DESIGN PROCEDURE

The design of a combinational circuit starts from the specification of the problem and culminates in a logic diagram or netlist that describes a logic diagram. The procedure involves the following steps:

1. **Specification:** Write a specification for the circuit if one is not already available.
2. **Formulation:** Derive the truth table or initial Boolean equations that define the required relationships between inputs and outputs.
3. **Optimization:** Apply two-level and multiple-level optimization. Draw a logic diagram or provide a netlist for the resulting circuit using ANDs, ORs, and inverters.
4. **Technology Mapping:** Transform the logic diagram or netlist to a new diagram or netlist using the available implementation technology.
5. **Verification:** Verify the correctness of the final design.

The specification can take a variety of forms such as text or an HDL description and should include the respective symbols or names for the inputs and outputs. Formulation converts the specification into forms that can be optimized. These forms are typically truth tables or Boolean expressions. It is important that verbal specifications be interpreted correctly when formulating truth tables or expressions. Often the specifications are incomplete, and any wrong interpretation may result in an incorrect truth table or expression. Optimization can be performed by any of a number available methods, such as algebraic manipulation, the K-map method, or computer-based simplification programs. In a particular application, specific criteria serve as a guide for choosing the optimization method. A practical design must consider constraints such as the cost of the gates used, maximum allowable propagation time of a signal through the circuit, and limitations on the fan-out of each gate. This is complicated by the fact that gate costs, gate delays, and fan-out limits are not known until the technology mapping stage. As a consequence, it is difficult to make a general statement about what constitutes an acceptable end result for optimization. In many cases, the optimization begins by satisfying an elementary objective, such as producing the simplified Boolean

expressions in a standard form for each output. The next step is multiple level optimization with terms shared between multiple outputs. In more sophisticated synthesis tools, optimization and technology mapping may be interspersed to improve the likelihood of meeting constraints. It may be necessary to repeat optimization and technology mapping multiple times to meet the specified constraints.

The remainder of this chapter illustrates the design procedure by using three examples. In the rest of this section, we perform the first three steps of design, specification, formulation, and optimization. We then consider implementation technologies and the final two steps in separate sections.

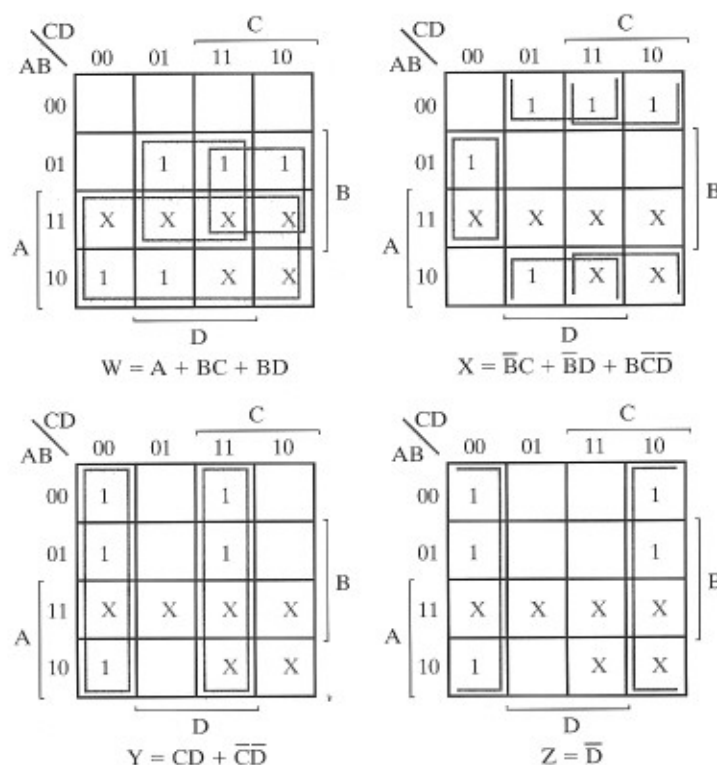
The first two example specifications are for a class of circuits called *code converters*, which translate information from one binary code to another. The inputs to the circuit are the bit combinations specified by the first code, and the outputs generate the corresponding bit combination of the second code. The combinational circuit performs the transformation from one code to the other. The first code converter example converts the BCD code to the excess-3 code for the decimal digits. The other converts the BCD code to the seven signals required to drive a seven-segment light-emitting diode (LED) display. The third example is the design of a 4-bit equality comparator that represents a circuit having a large number of inputs.

EXAMPLE 3-2 Design of a BCD-to-Excess-3 Code Converter

□ TABLE 3-1
Truth Table for Code Converter Example

Decimal Digit	Input BCD				Output Excess-3			
	A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	0	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	0	1	1
9	1	0	0	1	1	1	0	0

SPECIFICATION: The *excess-3 code* for a decimal digit is the binary combination corresponding to the decimal digit plus 3. For example, the excess-3 code for decimal digit 5 is the binary combination for $5 + 3 = 8$, which is 1000. The excess-3 code has desirable properties with respect to implementing decimal subtraction. Each BCD digit is four bits with the bits, from most significant to least significant, labeled A, B, C, D. Each excess-3 digit is four bits, with the bits, from most significant to least significant, labeled W, X, Y, Z.



□ **FIGURE 3-10**
Maps for BCD-to-Excess-3 Code Converter

FORMULATION: The excess-3 code word is easily obtained from a BCD code word by adding binary 0011 (3) to it. The resulting truth table relating the input and output variables is shown in Table 3-1. Note that the four BCD input variables may have 16 bit combinations, but only 10 are listed in the truth table. The six combinations 1010 through 1111 are not listed under the inputs, since these combinations have no meaning in the BCD code, and we can assume that they will never occur. Hence, for these input combinations, it does not matter what binary values we assign to the excess-3 outputs, and therefore, we can treat them as don't-care conditions.

OPTIMIZATION: Since this is a four-variable function, we use the K-maps in Figure 3-10 for the initial optimization of the four output functions. The maps are plotted to obtain simplified sum-of-products Boolean expressions for the outputs. Each of the four maps represents one of the outputs of the circuit as a function of the four inputs. The 1's in the maps are obtained directly from the truth table output columns. For example, the column under output W has 1's for minterms 5, 6, 7, 8, and 9. Therefore, the map for W must have 1's in the squares corresponding to these minterms. The six don't-care minterms, 10 through 15, are each marked with

an X in all the maps. The optimized functions are listed in sum-of-products form under the map for each output variable.

The two-level AND-OR logic diagram for the circuit can be obtained directly from the Boolean expressions derived from the maps. We apply multiple-level optimization as a second optimization step to determine if the gate input count, which is currently 26 (including inverters), can be reduced. In this optimization, we consider sharing subexpressions between the four output expressions. The following manipulation illustrates optimization with multiple-output circuits implemented with three levels of gates:

$$T_1 = C + D$$

$$W = A + BC + BD = A + BT_1$$

$$X = \overline{B}C + \overline{B}D + B\overline{C}\overline{D} = \overline{B}T_1 + B\overline{C}\overline{D}$$

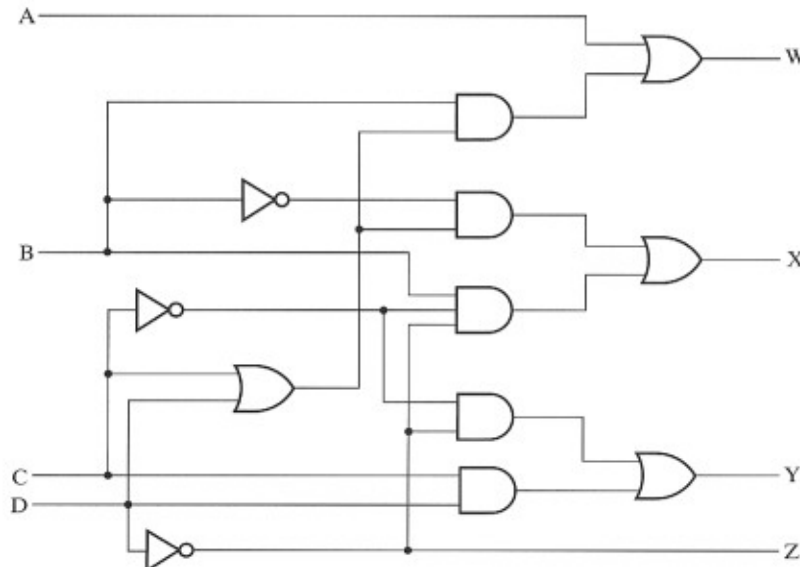
$$Y = CD + \overline{C}\overline{D}$$

$$Z = \overline{D}$$

The manipulation allows the gate producing $C + D$ to be shared by the logic for W and X , and reduced the gate input count from 26 to 22. This optimized result is viewed as being adequate and gives the logic diagram in Figure 3-11. ■

EXAMPLE 3-3 Design of a BCD-to-Seven-Segment Decoder

SPECIFICATION: Digital readouts found in many consumer electronic products such as alarm clocks often use Light Emitting Diodes (LEDs). Each digit of the readout



□ **FIGURE 3-11**
Logic Diagram of BCD-to-Excess-3 Code Converter



□ **FIGURE 3-12**
Seven-Segment Display

is formed from seven LED segments. Each segment can be illuminated by a digital signal. A BCD-to-seven-segment decoder is a combinational circuit that accepts a decimal digit in BCD and generates the appropriate outputs for the segments of the display for that decimal digit. The seven outputs of the decoder (a, b, c, d, e, f, g) select the corresponding segments in the display, as shown in Figure 3-12(a). The numeric designations chosen to represent the decimal digits are shown in Figure 3-12(b). The BCD-to-seven-segment decoder has four inputs, A, B, C , and D for the BCD digit and seven outputs, a through g , for controlling the segments.

FORMULATION: The truth table of the combinational circuit is listed in Table 3-2. On the basis of Figure 3-12(b), each BCD digit illuminates the proper segments for the decimal display. For example, BCD 0011 corresponds to decimal 3, which is displayed as segments a, b, c, d , and g . The truth table assumes that a logic 1 signal illuminates the segment and a logic 0 signal turns the segment off. Some seven-segment displays operate in reverse fashion and are illuminated by a logic 0 signal. For these displays, the seven outputs must be complemented. The six binary combinations 1010 through 1111 have no meaning in BCD. In the previous example, we assigned these combinations to don't-care conditions. If we do the same here, the design will

□ **TABLE 3-2**
Truth Table for BCD-to-Seven-Segment Decoder

BCD Input				Seven-Segment Decoder						
A	B	C	D	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
All other inputs				0	0	0	0	0	0	0

most likely produce some arbitrary and meaningless displays for the unused combinations. As long as these combinations do not occur, we can use that approach to reduce the complexity of the converter. A safer choice, turning off all the segments when any one of the unused input combinations occurs, avoids any spurious displays if any of the combinations occurs, but increases the converter complexity. This choice can be accomplished by assigning all 0's to minterms 10 through 15.

OPTIMIZATION: The information from the truth table can be transferred into seven K-maps from which the initial optimized output functions can be derived. The plotting of the seven functions in map form is left as an exercise. One possible way of simplifying the seven functions results in the following Boolean functions:

$$a = \overline{A}C + \overline{A}BD + \overline{B}C\overline{D} + A\overline{B}\overline{C}$$

$$b = \overline{A}\overline{B} + \overline{A}\overline{C}\overline{D} + \overline{A}CD + A\overline{B}\overline{C}$$

$$c = \overline{A}B + \overline{A}D + \overline{B}C\overline{D} + A\overline{B}\overline{C}$$

$$d = \overline{A}C\overline{D} + \overline{A}\overline{B}C + \overline{B}C\overline{D} + A\overline{B}\overline{C} + \overline{A}B\overline{C}D$$

$$e = \overline{A}C\overline{D} + \overline{B}C\overline{D}$$

$$f = \overline{A}B\overline{C} + \overline{A}\overline{C}\overline{D} + \overline{A}B\overline{D} + A\overline{B}\overline{C}$$

$$g = \overline{A}C\overline{D} + \overline{A}\overline{B}C + \overline{A}B\overline{C} + A\overline{B}\overline{C}$$

Independent implementation of these seven functions requires 27 AND gates and 7 OR gates. However, by sharing the six product terms common to the different output expressions, the number of AND gates can be reduced to 14 along with a substantial savings in gate input count. For example, the term $\overline{B}C\overline{D}$ occurs in a , c , d , and e . The output of the AND gate that implements this product term goes directly to the inputs of the OR gates in all four functions. For this function, we stop optimization with the two-level circuit and shared AND gates, realizing that it might be possible to reduce the gate input count even further by applying multiple level optimization. ■

The BCD-to-seven-segment decoder is called a decoder by most manufacturers of integrated circuits because it decodes a binary code for a decimal digit. However, it is actually a code converter that converts a four-bit decimal code to a seven-bit code. The word “decoder” is usually reserved for another type of circuit, presented in the next chapter.

In general, the total number of gates can be reduced in a multiple-output combinational circuit by using common terms of the output functions. The maps of the output functions may help find the common terms by finding identical implicants from two or more maps. Some of the common terms may not be prime implicants of the individual functions. The designer must be inventive and combine squares in the maps in such a way as to create common terms. This can be done more formally by using a procedure for simplifying multiple-output functions. The prime implicants are defined not only for each individual function, but

also for all possible combinations of the output functions. These prime implicants are formed by using the AND operator on every possible nonempty subset of the output functions and finding the prime implicants of each of the results. Using this entire set of prime implicants, a formal selection process can be used to find the optimum two-level multiple output circuit. Such a procedure is implemented in various forms in logic simplification software within logic synthesis tools and is the method used to obtain the equations in Example 3-3.

EXAMPLE 3-4 Design of a 4-bit Equality Comparator

SPECIFICATION: The inputs to the circuit consist of two vectors: A(3:0) and B(3:0). Vector A consists of four bits, A(3), A(2), A(1), and A(0), with A(3) as the most significant bit. Vector B has a similar description with B replaced by A. The output of the circuit is a single bit variable E. Output E is equal to 1 if A and B are equal and equal to 0 if A and B are unequal.

FORMULATION: Since this circuit has eight inputs, use of a truth table for formulation is impractical. In order for A and B to be equal, the bit values in each of the respective positions, 3 down to 0, of A and B must be equal. If all of the bit positions for A and B contain equal values in every position, then $E = 1$; otherwise, $E = 0$.

OPTIMIZATION: For this circuit, we use intuition to immediately develop a multiple level circuit using hierarchy. Since comparison of a bit from A and the corresponding bit from B must be done in each of the bit positions, we can decompose the problem into four 1-bit comparison circuits and an additional circuit that combines the four comparison circuit outputs to obtain E. For bit position i, we define the circuit output E_i to be 0 if A_i and B_i have the same values and $E_i = 1$ if A_i and B_i have different values. This circuit can be described by the equation

$$E_i = \bar{A}_i B_i + A_i \bar{B}_i$$

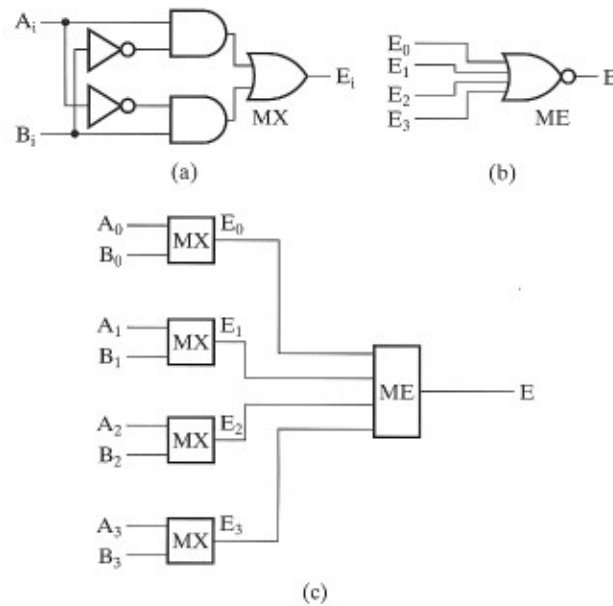
which has the circuit diagram shown in Figure 3-13(a). By using hierarchy and reuse, we can employ four copies of this circuit, one for each of the four bits of A and B. Output $E = 1$ only if all of the E_i values are 0. This can be described by the equation

$$E = \overline{E_0 + E_1 + E_2 + E_3}$$

and has the diagram given in Figure 3-13(b). Both of the circuits given are optimum two-level circuits. The overall circuit can be described hierarchically by the diagram in Figure 3-13(c). ■

3-4 TECHNOLOGY MAPPING

There are three primary ways of designing VLSI circuits. In *full custom design*, an entire design of the chip, down to the smallest detail of the layout, is performed.



□ **FIGURE 3-13**
Hierarchical Diagram for a 4-bit Equality Comparator

Since this process is very expensive, custom design can be justified only for dense, fast ICs that are likely to be sold in sizable quantities.

A closely related technique is *standard cell design*, in which large parts of the design have been performed ahead of time or, possibly, used in previous designs. The predesigned parts are connected to form the IC design. This intermediate-cost methodology gives lower density and lower performance than full custom design.

The third approach to VLSI design is the use of a *gate array*. A gate array uses a rectangular pattern of gates fabricated in silicon. This pattern is repeated thousands of times, so that the entire chip contains identical gates. Depending on the technology used, pattern arrays of 1000 to millions of gates can be fabricated within a single IC. The application of a gate array requires that the design specify how the gates are interconnected and how the interconnections are routed. Many steps of the fabrication process are common and independent of the final logic function. These steps are economical, since they can be used for numerous different designs. In order to customize the gate array to the particular design, additional fabrication steps are required to interconnect the gates. Due to the commonality of fabrication steps and ability to share the results of these steps with many different designs, this is the lowest cost method among the fixed implementation technologies.

For standard cell and gate array technologies, circuits are constructed by interconnecting cells. The collection of cells available for a given implementation technology is called a *cell library*. In order to design in terms of a cell

library, it is necessary to characterize each of the cells (i.e., provide a detailed specification of the cells for use by the designer). A library of characterized cells provides a foundation for the technology mapping of circuits. Coupled with the library is a technology mapping procedure. In this section, we consider technology mapping procedures for cell libraries consisting of (1) single gate types such as NAND gates, and (2) multiple gate types. Technology mapping may focus on a number of the dimensions of the design space, particularly on cost and performance. For simplicity, our procedures focus only on optimizing cost. Further, these procedures are rudimentary versions of technology mapping algorithms used in computer-aided design tools and are suitable for manual application to only the simplest of circuits. Nevertheless, they give us some insight into how a design using AND gates, OR gates, and inverters can be transformed into cost-effective designs using cell types supported by available implementation technologies.

Cell Specification

Specifications for cells used in standard cell and gate array designs typically have many components. Typical components include the following:

1. A schematic or logic diagram for the function of the cell.
2. A specification of the area the cell occupies, often normalized to the area of a small cell such as that of a minimum area inverter.
3. The input loading, in standard loads, that each input of a cell presents to the output driving it.
4. Delays from each input of a cell to each output of a cell (if a path from the input to output exists), including the effect of the number of standard loads driven by the output.
5. One or more templates for the cell for use in performing technology mapping.
6. One or more HDL models for the cell.

If the tools used provide automated layout, then the following additional components are also included in the specification:

7. An integrated circuit layout for the cell.
8. A floorplan layout showing the locations of the inputs, outputs and power and ground connections for the cell for use during the cell interconnection process.

The first five components listed are included in a simple technology library of cells in the next subsection. Some of these components are discussed in more detail.

Libraries

The cells for a particular design technology are organized into one or more libraries. A *library* is a collection of cell specifications. A circuit that initially consists of

AND, OR and NOT gates is converted by technology mapping to one that uses only cells from the applicable libraries. A very small technology library is described in Table 3-3. This library contains primitive inverting gates with fan-ins up to four and a single AOI circuit.

The first column of the table contains a descriptive name for the cell and the second column contains the cell schematic. The third column contains the area of the cell normalized to the area of a minimum inverter. Area can be used as a very simple measure of the cost of the cell. The next column gives the typical load that a cell input places on the gate driving it. The load values are normalized to a quantity called a standard load which in this case is the capacitance presented to the driving circuit by the input of an inverter. In the case of the cells given, the input loads are almost all the same. The fifth column gives a simple linear equation for calculating the typical input-to-output delay for the cell. The variable SL is the sum of all of the standard loads presented by the inputs of cells driven by the cell output. It may also contain an estimate, in standard loads, of the capacitance of the wiring connecting the cell output to the inputs of other cells. This equation illustrates the notion that cell delays consist of some fixed delay, plus a delay that is dependent upon the capacitance loading of the cell as represented by SL . Cell delay calculation is illustrated in Example 3-5.

EXAMPLE 3-5 Calculation of Cell Delay

This example illustrates the effect of loading on cell delay. A 2NAND output drives the following cells: an inverter, a 4NAND, and a 4NOR. The sum of the standard loads in this case is

$$SL = 1.00 + 0.95 + 0.80 = 2.75$$

With this value, the delay of the 2NAND driving the cells specified is






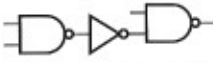

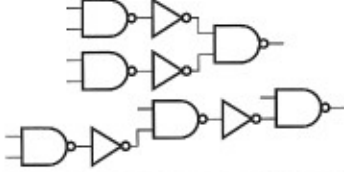

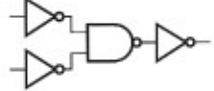

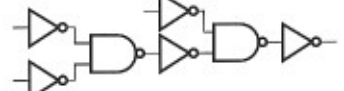

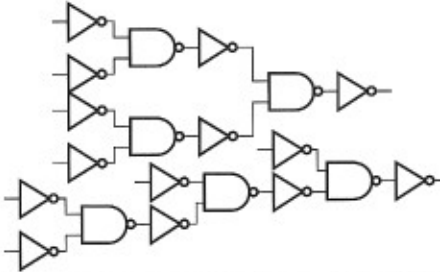

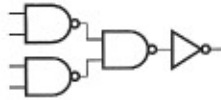
$$t_p = 0.05 + 0.014 \times 2.75 = 0.089 \text{ ns}$$

The final column of the table gives templates for the cell function that use only basic functions as components. In this case, the basic functions are a 2-input NAND gate and an inverter. Use of these basic function templates provides a way of representing each cell function in a “standard” form. As illustrated by the 4-input NAND and NOR cells, the basic function template for a cell is not necessarily unique. It should be noted that these diagrams represent only a netlist, not actual location, orientation, or interconnect layout. For example, consider the template for the 3NAND. If the left NAND and the following inverter were connected to the top input of the right NAND, instead of to its bottom input, the template would be unchanged. The value of these templates will become apparent in the next section on mapping techniques.

Mapping Techniques

In this subsection, we consider the mapping process for fixed cell technologies. A convenient way to implement a Boolean function with NAND gates is to obtain the optimized Boolean function in terms of the Boolean operators AND, OR, and

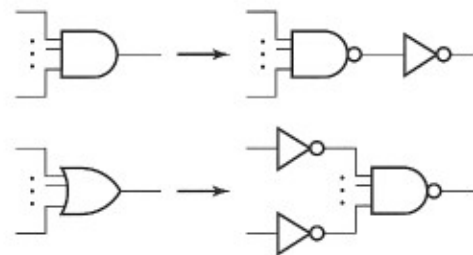
□ TABLE 3-3
Example Cell Library for Technology Mapping

Cell Name	Cell Schematic	Normalized Area	Typical Input Load	Typical Input-to-Output Delay	Basic Function Templates
Inverter		1.00	1.00	$0.04 + 0.012 \times SL$	
2NAND		1.25	1.00	$0.05 + 0.014 \times SL$	
3NAND		1.50	1.00	$0.06 + 0.017 \times SL$	
4NAND		2.00	0.95	$0.07 + 0.021 \times SL$	
2NOR		1.25	1.00	$0.06 + 0.018 \times SL$	
3NOR		2.00	0.95	$0.15 + 0.012 \times SL$	
4NOR		3.25	0.80	$0.17 + 0.012 \times SL$	
2-2 AOI		2.25	0.95	$0.07 + 0.019 \times SL$	

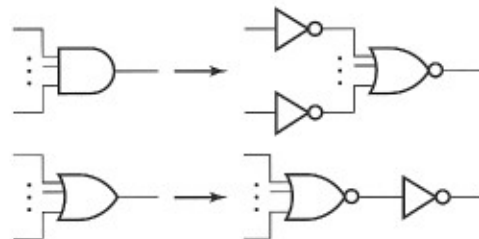
NOT and then map the function to NAND logic. The conversion of an algebraic expression from AND, OR, and NOT to NAND can be done by a simple procedure that changes AND-OR logic diagrams to NAND logic diagrams. The same conversion applies for NOR gates.

Given an optimized circuit that consists of AND gates, OR gates, and inverters, the following procedure produces a circuit using either NAND (or NOR) gates with unrestricted gate fan-in:

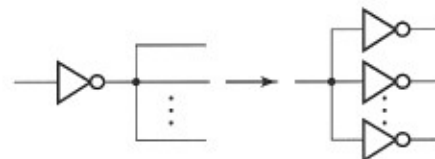
1. Replace each AND and OR gate with the NAND (NOR) gate and inverter equivalent circuits shown in Figure 3-14 (a) and (b).
2. Cancel all inverter pairs.
3. Without changing the logic function, (a) “push” all inverters lying between (i) either a circuit input or a driving NAND(NOR) gate output and (ii) the



(a) Mapping to NAND gates



(b) Mapping to NOR gates



(c) Pushing an inverter through a “dot”



(d) Cancelling inverter pairs

□ **FIGURE 3-14**

Mapping of AND Gates, OR Gates and Inverters to NAND gates, NOR gates, and Inverters

driven NAND(NOR) gate inputs toward the driven NAND(NOR) gate inputs. Cancel pairs of inverters in series whenever possible during this step. (b) Replace inverters in parallel with a single inverter that drives all of the outputs of the parallel inverters. (c) Repeat (a) and (b) until there is at most one inverter between the circuit input or driving NAND(NOR) gate output and the attached NAND(NOR) gate inputs.

In Figure 3-14 (c), the rule for pushing an inverter through a “dot” is given. The inverter on the input line to the dot is replaced with inverters on each of the output lines from the dot. The cancellation of pairs of inverters in Figure 3-14(d) is based on the Boolean algebraic identity

$$\overline{\overline{X}} = X$$

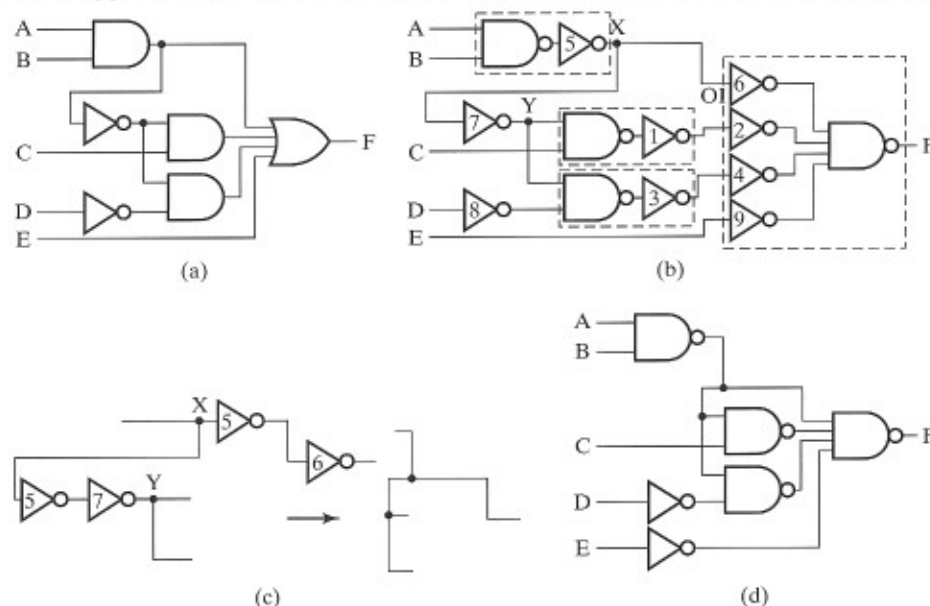
The next example illustrates this procedure for NAND gates.

EXAMPLE 3-6 Implementation with NAND Gates

Implement the following optimized function with NAND gates:

$$F = AB + (\overline{A}\overline{B})C + (\overline{A}\overline{B})\overline{D} + E$$

The AND, OR, inverter implementation is given in Figure 3-15(a). In Figure 3-15 (b), step 1 of the procedure has been applied replacing each AND gate and OR gate with its equivalent circuit using NAND gates and inverters from Figure 3-14(a). Labels appear on dots and inverters to assist in the explanation. In step 2, the



□ **FIGURE 3-15**
Solution to Example 3-6

inverter pairs (1, 2) and (3, 4), cancel, giving direct connections between the corresponding NAND gates in Figure 3-15(d). As shown in Figure 3-15(c), inverter 5 is pushed through X and cancels with inverters 6 and 7, respectively. This gives direct connections between the corresponding NAND gates in Figure 3-15(d). No further steps can be applied since inverters 8 and 9 cannot be paired with other inverters, and must remain in the final mapped circuit in Figure 3-15(d). The next example illustrates this procedure for NOR gates. ■

EXAMPLE 3-7 Implementation with NOR Gates

Implement the same optimized Boolean function used in Example 3-7 with NOR gates:

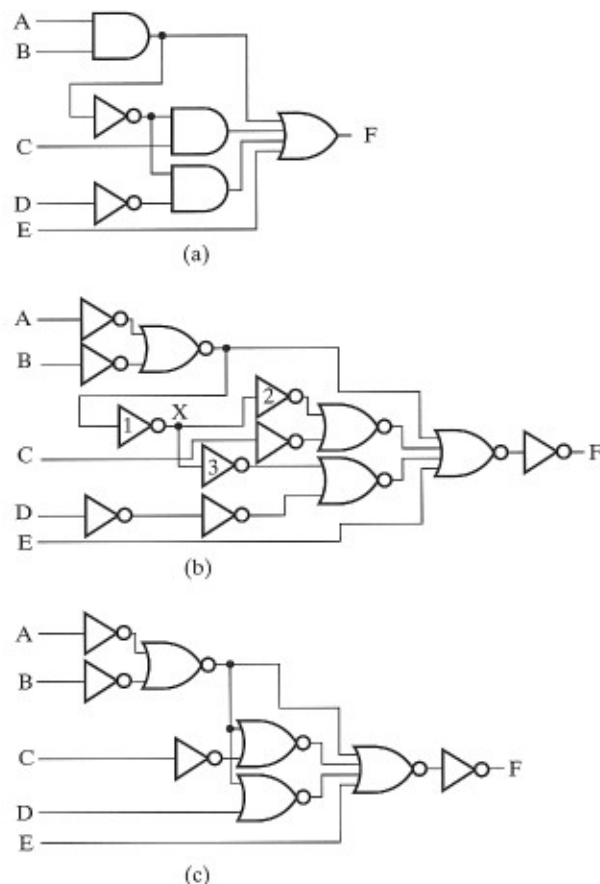
$$F = AB + (\overline{A}B)C + (\overline{A}B)\overline{D} + E$$

The AND, OR, inverter implementation is given in Figure 3-16(a). In Figure 3-16(b), step 1 of the procedure has been applied replacing each AND gate and OR gate with its equivalent circuit using NOR gates and inverters from Figure 3-14(b). Labels appear on dots and inverters to assist in the explanation. In step 2, inverter 1 can be pushed through dot X to cancel with inverters 2 and 3, respectively. The pair of inverters on the D input line cancel as well. The single inverters on input lines A , B , and C and output line F must remain, giving the final mapped circuit that appears in Figure 3-16(c). ■

In Example 3-6, the gate input cost of the mapped circuit is 12, and, in Example 3-7, the gate input cost is 14, so the NAND implementation is less costly. Also, the NAND implementation involves a maximum of three gates in series while the NOR implementation has a maximum of five gates in series. Due to the longer series of gates in the NOR circuit, the maximum delay from an input change to a corresponding output change is likely to be longer.

In the preceding procedure and examples, the mapping target consisted of a single gate type, either NAND gates or NOR gates. The following procedure handles multiple gate types:

1. Replace each AND and OR gate with an optimum equivalent circuit consisting only of 2-input NAND gates and inverters.
2. In each line in the circuit attached to a circuit input, a NAND gate input, a NAND gate output, or a circuit output in which no inverter appears, insert a serial pair of inverters.
3. Perform a replacement of connections of NAND gates and inverters by the available library cells such that the gate input cost which results within fan-out free subcircuits is optimized. A *fan-out free subcircuit* is a circuit in which each gate output drives a single gate input. (This step is not covered here in detail due to its complexity, but is provided on the course website with an example. The templates shown in the right column of Table 3-3 are used to match connections of NAND gates and inverters to available library cells.)



□ **FIGURE 3-16**
Solution to Example 3-7

4. (a) Without changing the logic function, “push” all inverters, lying between (i) a circuit input or a driving gate output and (ii) the driven gate inputs, toward the driven gate inputs. Cancel pairs of inverters in series whenever possible during this step. (b) Replace inverters in parallel with a single inverter that drives all of the outputs of the parallel inverters. (c) Repeat (a) and (b) until there is at most one inverter between the circuit input or driving gate output and the attached driven gate inputs.

This procedure is one of the foundations for technology mapping in commercial synthesis tools. The intermediate replacement of the initial circuit gates with only 2-input NAND gates and inverters breaks the circuit up into small pieces in order to provide the maximum flexibility in mapping cells to achieve an optimized result. Example 3-8 shows an implementation approach using a small cell library.

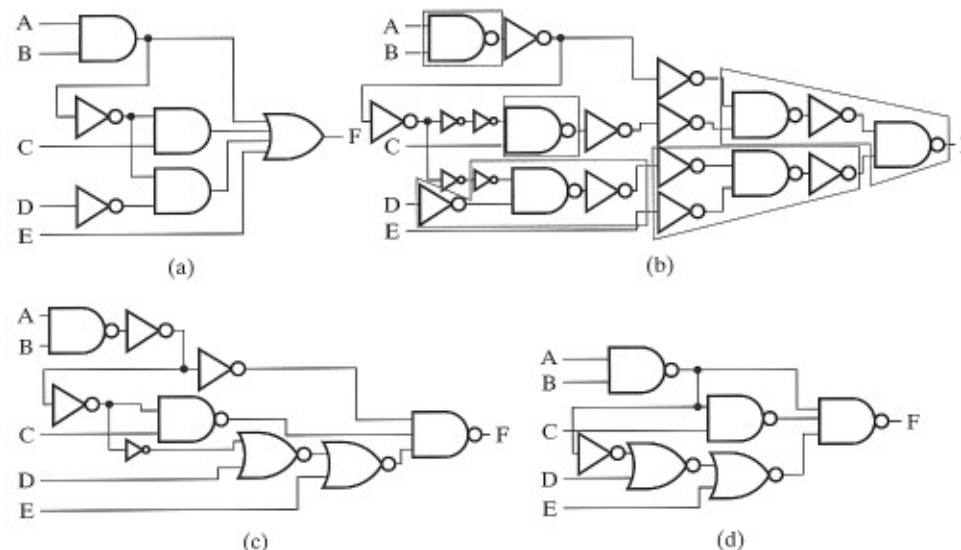
EXAMPLE 3-8 Implementation with a Small Cell Library

Implement the same optimized Boolean function used in Examples 3-6 and 3-7

$$F = AB + (\overline{AB})C + (\overline{AB})\overline{D} + E$$

with a cell library containing a 2-input NAND gate, 3-input NAND gate, a 2-input NOR gate, and an inverter. The AND, OR, inverter implementation is given in Figure 3-17(a). In Figure 3-17(b), steps 1 and 2 of the procedure have been applied. Each AND gate and each OR gate has been replaced with its equivalent circuit made up of 2-input NAND gates and inverters. Pairs of inverters have been added to the internal lines without inverters. Due to lack of space, the pairs of inverters on the inputs and outputs are not shown. Application of step 3 results in the mapping to the cells from the cell library shown in Figure 3-17(c). The blue outlines enclose connections of NAND gates and inverters, each of which is to be replaced by an available cell using the templates in Table 3-3. In this case, all of the available cells have been used at least once. Application of step 4 cancels out three of the inverters, giving the final mapped circuit in Figure 3-17(d).

The solution for Example 3-8 has a gate input cost of 12, compared to costs of 12 and 14 for Examples 3-6 and 3-7, respectively. Although the cost in Examples 3-6 and 3-8 are identical, it should be noted that the cell libraries differ. In particular, Example 3-6 benefits from the use of a 4-input NAND gate that is unavailable in Example 3-8. Without this cell, the solution would cost two additional gate inputs. Thus, the use of a more diverse cell library has provided a cost benefit.



□ **FIGURE 3-17**
Solution to Example 3-8

To provide continuity with examples in earlier sections of this chapter, the following example shows the mapping of the BCD-to-Excess-3 Code Converter for an expanded cell library.

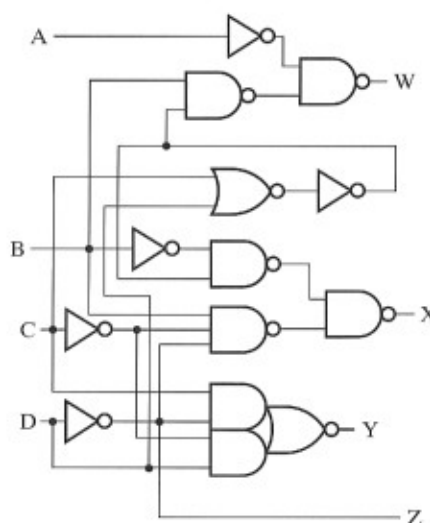
EXAMPLE 3-9 Technology Mapping for BCD-to-Excess-3 Code Converter

The final result of the technology mapping for the BCD-to-Excess-3 Code Converter is given in Figure 3-18. The original AND, OR, inverter logic diagram appears in Figure 3-11, and the cell library used is given in Table 3-3. The optimization has resulted in the use of the following cells from that library: inverters, 2-input NANDs, a 2-input NOR, and a 2-2 AOI.

The gate input cost of the mapped circuit in Example 3-9 is 22, with a gate input cost of 22 for the original AND, OR, NOT circuit. The optimization procedure, aside from locally minimizing inverters, works separately on the various parts of the circuit. These parts are separated by gate fan-outs in the original AND-OR circuit. The selection of these points in the optimization can effect the optimality of the final result. In the case of this circuit, a different original circuit may yield a better optimization. In general, this problem of separate optimization and mapping is handled by using combined optimization steps and mapping steps in commercial logic optimization tools.



ADVANCED TECHNOLOGY MAPPING This supplement on technology mapping, including detailed examples illustrating the mapping procedure for general cell libraries, is available on the Companion Website for the text.



□ **FIGURE 3-18**

Technology Mapping Example: BCD-to-Excess-3 Code Converter

3-5 VERIFICATION

In this section, manual logic analysis and computer simulation-based logic analysis are considered, both with the goal of verification of circuit function (i.e., determination of whether or not a given circuit implements its specified function). If the circuit does not meet its specification, then it is incorrect. As a consequence, verification plays a vital role in preventing incorrect circuit designs from being manufactured and used. Logic analysis also can be used for other purposes, including redesign of a circuit and determination of the function of a circuit.

In order to verify a combinational circuit, it is essential that the specification be unambiguous and correct. As a consequence, specifications such as truth tables, Boolean equations, and HDL code are most useful. Initially, we examine manual verification by continuing with the design examples we introduced in this chapter.

Manual Logic Analysis

Manual logic analysis consists of finding Boolean equations for the circuit outputs or, additionally, finding the truth table for the circuit. The approach taken here emphasizes finding the equations and then using those equations to find the truth table. In finding the equation for a circuit, it is often convenient to break up the circuit into subcircuits by defining intermediate variables at selected points in the circuit. Points typically selected are those at which a gate output drives two or more gate inputs. Such points are often referred to as *fan-out points*. Fan-out points from a single inverter on an input typically would not be selected. The determination of logic equations for a circuit is illustrated using the BCD-to-Excess-3 Code Converter circuit designed in previous sections.

EXAMPLE 3-10 Manual Verification of BCD-to-Excess-3 Code Converter

Figure 3-19 shows (a) the original truth table specification, (b) the final circuit implementation, and (c) an incomplete truth table to be completed from the implementation and then compared to the original truth table. The truth table values are to be determined from Boolean equations for W , X , Y , and Z derived from the circuit. The point $T1$ is selected as an intermediate variable to simplify the analysis:

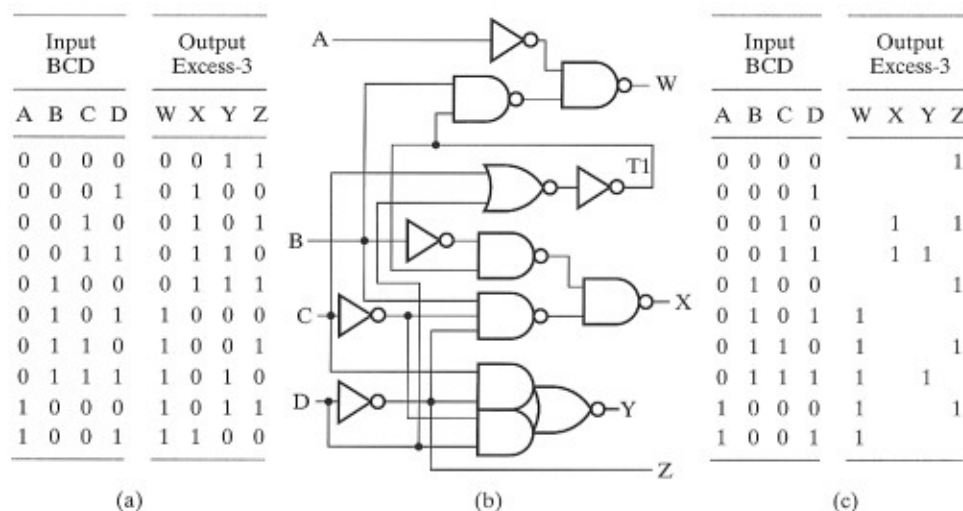
$$T1 = \overline{\overline{C + D}} = C + D$$

$$W = \overline{\overline{A \cdot (T1 \cdot B)}} = A + B \cdot T1$$

$$X = \overline{(\overline{B \cdot T1}) \cdot (\overline{B \cdot \overline{C} \cdot \overline{D}})} = \overline{B} \cdot T1 + B\overline{C} \cdot \overline{D}$$

$$Y = \overline{\overline{CD} + \overline{CD}} = CD + \overline{CD}$$

$$Z = \overline{D}$$



□ **FIGURE 3-19**
Verification: BCD-to-Excess-3 Code Converter

Substituting the expression for $T1$ in the equations for W and X , we have

$$W = A + B(C + D) = A + BC + BD$$

$$X = \bar{B}(C + D) + B\bar{C}\bar{D} = \bar{B}C + \bar{B}D + B\bar{C}\bar{D}$$

Each of the product terms in the four output equations can be mapped to 1's in the truth table in Figure 3-19 (c). The mappings of the 1's for A , $\bar{B}C$, BD , CD , and \bar{D} are shown. After the remaining product terms are mapped to 1's, the blank entries are filled with 0's. The new truth table in this case will match the original one, verifying that the circuit is correct. ■



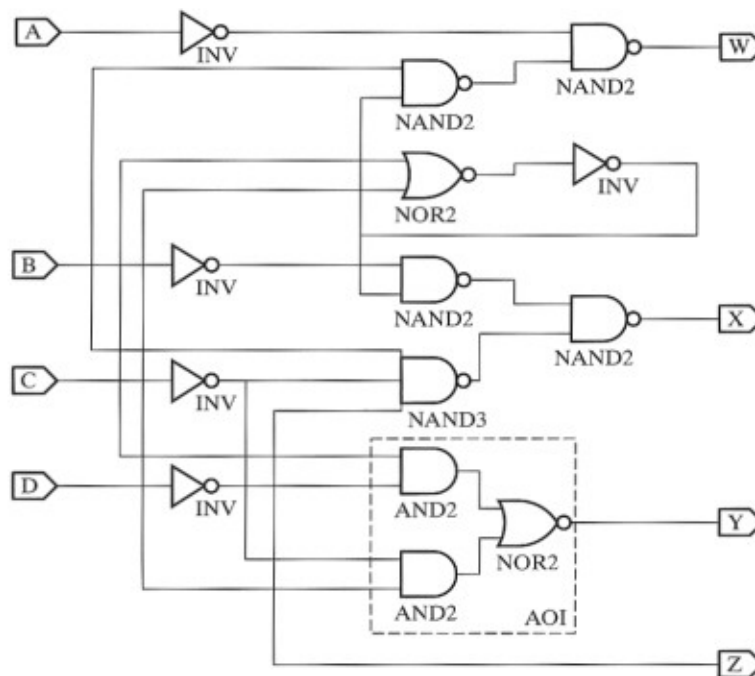
LOGIC ANALYSIS This supplement, including additional logic analysis techniques and examples, is available on the Companion Website for the text.

Simulation

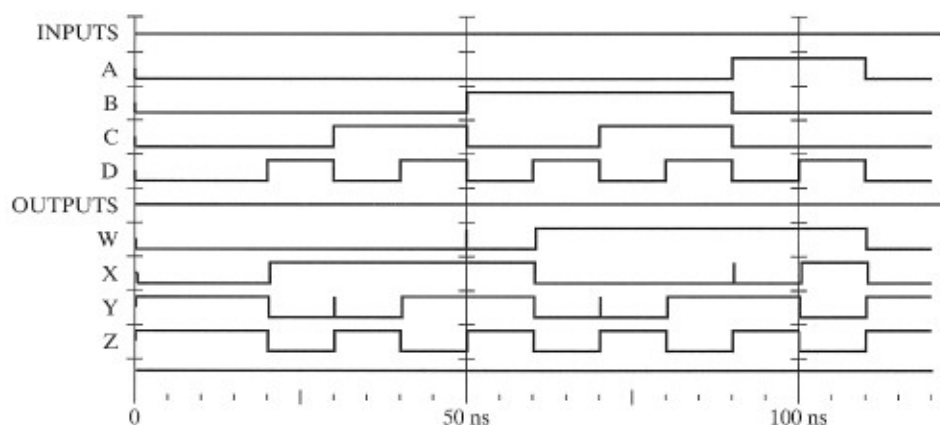
An alternative to manual verification is the use of computer simulation for verification. Using a computer permits truth table verification to be done for a significantly larger number of variables and greatly reduces the tedious analysis effort required. Since simulation uses applied values, if possible, it is desirable for thorough verification to apply all possible input combinations. The next example illustrates the use of Xilinx ISE4.2i FPGA development tools and XE II Modelsim simulator to verify the BCD-to-Excess-3 Code Converter using all possible input combinations from the truth table.

EXAMPLE 3-11 Simulation-based Verification of BCD-to-Excess-3 Code Converter

Figure 3-19 shows (a) the original truth table specification, and (b) the final circuit implementation of the BCD-to-Excess 3 Code Converter. The circuit implementation has been entered into Xilinx ISE 4.2i as the schematic shown in Figure 3-20. Since there are no AOIs available in the symbol library, the AOI has been modeled with available symbols. In addition to entering the schematic, the input combinations given in Figure 3-19(a) have also been entered as a waveform. These input waveforms are given in the INPUTS section of the simulation output shown in Figure 3-21. The simulation of the input waveforms applied to the circuit produces the output waveforms given in the OUTPUT section. Examining each input combination and the corresponding output combination represented by the waveforms, we can manually verify whether the outputs match the original truth table. Beginning with $(A,B,C,D) = (0,0,0,0)$ in the input waveform, we find that the corresponding output waveform represents $(W,X,Y,Z) = (0,0,1,1)$. Continuing, for $(A,B,C,D) = (0,0,0,1)$, the values for the output waveforms are $(W,X,Y,Z) = (0,1,0,0)$. In both cases, the values are correct. This process of checking the waveform values against the specification can be continued for the remaining eight input combinations to complete the verification. ■



□ **FIGURE 3-20**
Schematic for Simulation of BCD-to-Excess-3 code Converter



□ FIGURE 3-21

Example 3-10: Simulation Results—BCD-to-Excess-3 Code Converter



ADVANCED VERIFICATION This supplement, containing additional verification techniques and examples, is available on the Companion Website for the text.

3-6 PROGRAMMABLE IMPLEMENTATION TECHNOLOGIES

Thus far, we have introduced implementation technologies that are fixed in the sense that they are fabricated as integrated circuits or by connecting together integrated circuits. In contrast, programmable logic devices (PLDs) are fabricated with structures that implement logic functions and structures that are used to control connections or to store information specifying the actual logic functions implemented. These latter structures require *programming*, a hardware procedure that determines which functions are implemented. The next three subsections deal with three types of simple programmable logic devices (PLDs): the read-only memory (ROM), the programmable logic array (PLA), and the programmable array logic (PAL[®]) device. In a Companion Website supplement, the more complex field-programmable gate array (FPGA) is discussed and illustrated. Before treating PLDs, we deal with the supporting programming technologies. In PLDs, programming technologies are applied to (1) establish or break interconnections, (2) build lookup tables, and (3) control transistor switching. We will relate the technologies to these three applications.

The oldest of programming technologies for controlling connections is the use of *fuses*. Each of the programmable points in the PLD consists of a connection formed by a fuse. When a voltage considerably higher than the normal power supply voltage is applied across the fuse, the high current breaks the connection by “blowing out” the fuse. The two connection states, CLOSED and OPEN, are represented by an intact and blown fuse, respectively.

A second programming technology for controlling connections is *mask programming*, which is done by the semiconductor manufacturer during the last steps of the chip fabrication process. Connections are made or not made in the metal layers serving as conductors in the chip. Depending on the desired function for the

chip, the structure of these layers is determined by the fabrication process. The procedure is costly because masks for generating the layers and custom fabrication are required for each customer. For this reason, mask programming is economical only if a large quantity of the same PLD configuration is ordered.

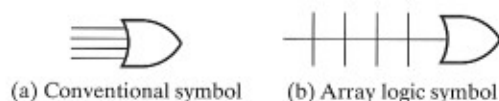
A third programming technology for controlling connections is the use of *antifuse*. As the name suggests, the antifuse is just the opposite of a fuse. In contrast to a fuse, an antifuse consists of a small area in which two conductors are separated by a material having a high resistance. The antifuse acts as an OPEN path before programming. By applying a voltage somewhat higher than the normal power supply voltage across the two conductors, the material separating the two conductors is melted or otherwise changed to a low resistance. The low-resistance material conducts, causing a connection (i.e., a CLOSED path) to be formed.

All three of the preceding connection technologies are permanent. The devices cannot be reprogrammed, because irreversible physical changes have occurred as a result of device programming. Thus, if the programming is incorrect or needs to be changed, the device must be discarded.

The final programming technology that can be applied for controlling connections is a single-bit storage element driving the gate of an MOS *n*-channel transistor at the programming point. If the stored bit value is a 1, then the transistor is turned ON, and the connection between its source and drain forms a CLOSED path. For the stored bit value equal to 0, the transistor is OFF, and the connection between its source and drain is an OPEN path. Since storage element content can be changed electronically, the device can be easily reprogrammed. But in order to store values, the power supply must be available. Thus, the storage element technology is volatile; that is, the programmed logic is lost in the absence of the power supply voltage.

The second application of programming technologies is building lookup tables. In addition to controlling connections, storage elements are ideal for building these tables. In this case, the input combination for the truth table is used to select a storage element containing the corresponding output value for the truth table and provide it as the logic function output. The hardware consists of (1) the storage elements, (2) the hardware to program values into the storage elements, and (3) the logic that selects the storage element contents to be applied to the logic function output. Because storage elements are being selected by the input values, the storage elements combined with the hardware in (3) resemble a memory in which stored data values are selected to appear on the memory output by using an address applied to the inputs. Thus, the logic can be implemented simply by storing the truth table in the memory—hence the term *lookup table*.

The third application of programming technologies is control of transistor switching. The most popular technology is based on storing charge on a floating gate. The latter is located below the regular gate within an MOS transistor and is completely isolated by an insulating dielectric. Stored negative charge on the floating gate makes the transistor impossible to turn ON. The absence of stored negative charge makes it possible for the transistor to turn ON if a HIGH is applied to its regular gate. Since it is possible to add or remove the stored charge, these technologies can permit erasure and reprogramming.

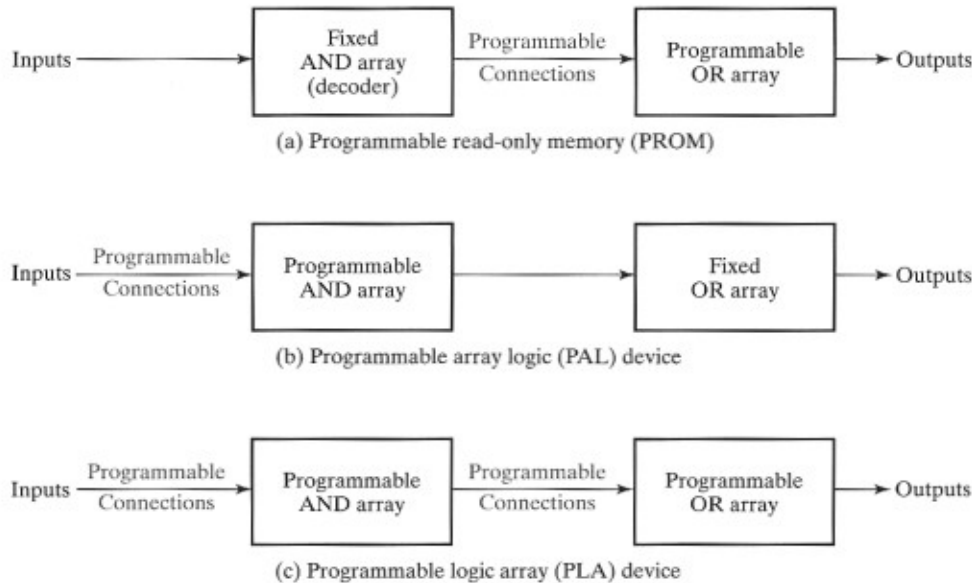


□ **FIGURE 3-22**
Conventional and Array Logic Symbols for OR Gate

Two technologies using control of transistor switching are called *erasable* and *electrically erasable*. Programming applies combinations of voltage higher than normal power supply voltages to the transistor. Erasure uses exposure to a strong ultraviolet light source for a specified amount of time. Once this type of chip has been erased, it can be reprogrammed. An electrically erasable device can be erased by a process somewhat similar to the programming process, using voltages higher than the normal power supply value. Since transistor control prevents or allows a connection to be established between the source and the drain, it is really a form of connection control, giving a choice between (1) always OPEN or (2) OPEN or CLOSED, depending on an applied HIGH or LOW, respectively, on the regular transistor gate. A third technology based on control of transistor switching is *flash* technology, which is very widely used in *flash memories*. Flash technology is a form of electrically-erasable technology that has a variety of erase options including the erase of stored charge from individual floating gates, all of the floating gates, or specific subsets of floating gates.

A typical PLD may have hundreds to millions of gates. Some, but not all, programmable logic technologies have high fan-in gates. In order to show the internal logic diagram for such technologies in a concise form, it is necessary to employ a special gate symbology applicable to array logic. Figure 3-22 shows the conventional and array logic symbols for a multiple-input OR gate. Instead of having multiple input lines to the gate, we draw a single line to the gate. The input lines are drawn perpendicular to this line and are selectively connected to the gate. If an *x* is present at the intersection of two lines, there is a connection. If an *x* is not present, then there is no connection. In a similar fashion, we can draw the array logic for an AND gate. Since this was first done for a fuse-based technology, the graphics representation, when marked with the selected connections, is referred to as a *fuse map*. We will use the same graphics representation and terminology even when the programming technology is not fuses. This type of graphics representation for the inputs of gates will be used subsequently in drawing logic diagrams.

We next consider three distinct programmable device structures. We will describe each of the structures and indicate which of the technologies is typically used in its implementation. These types of PLDs differ in the placement of programmable connections in the AND-OR array. Figure 3-23 shows the locations of the connections for the three types. Programmable read-only memory (PROM) as well as flash memory has a fixed AND array constructed as a decoder and programmable connections for the output OR gates. The PROM implements Boolean functions in sum-of-minterms form. The programmable array logic (PAL[®]) device has a programmable connection AND array and a fixed OR array. The AND gates are programmed to provide the product terms for the Boolean functions, which are



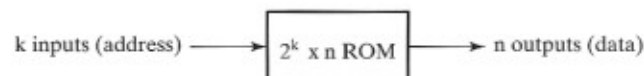
□ **FIGURE 3-23**
Basic Configuration of Three PLDs

logically summed in each OR gate. The most flexible of the three types of PLD is the programmable logic array (PLA), which has programmable connections for both AND and OR arrays. The product terms in the AND array may be shared by any OR gate to provide the required sum-of-products implementation. The names PLA and PAL[®] emerged for devices from different vendors during the development of PLDs.

Read-Only Memory

A read-only memory (ROM) is essentially a device in which “permanent” binary information is stored. The information must be specified by the designer and is then embedded into the ROM to form the required interconnection or electronic device pattern. Once the pattern is established, it stays within the ROM even when power is turned off and on again; that is, ROM is nonvolatile.

A block diagram of a ROM device is shown in Figure 3-24. There are k inputs and n outputs. The inputs provide the address for the memory, and the outputs give

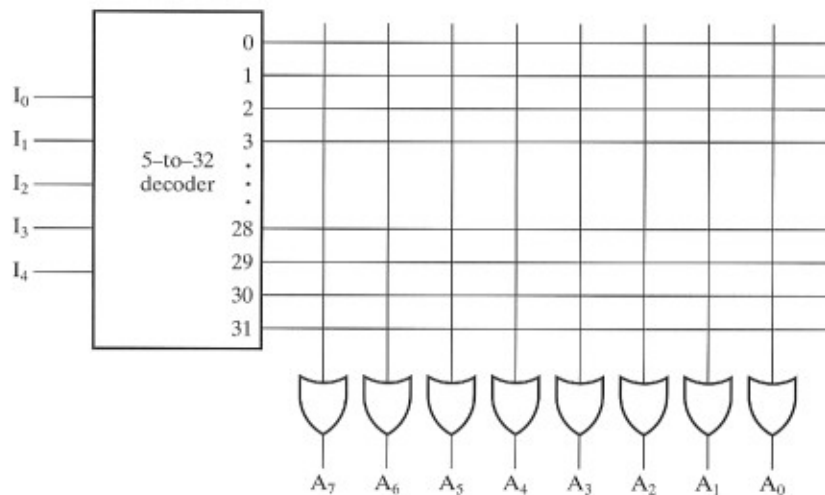


□ **FIGURE 3-24**
Block Diagram of ROM

the data bits of the stored word that is selected by the address. The number of words in a ROM device is determined from the fact that k address input lines can specify 2^k words. Note that ROM does not have data inputs, because it does not have a write operation. Integrated circuit ROM chips have one or more enable inputs and come with three-state outputs to facilitate the construction of large arrays of ROM.

Consider, for example, a 32×8 ROM. The unit consists of 32 words of 8 bits each. There are five input lines that form the binary numbers from 0 through 31 for the address. Figure 3-25 shows the internal logic construction of this ROM. The five inputs are decoded into 32 distinct outputs by means of a 5-to-32-line decoder. Each output of the decoder represents a memory address. The 32 outputs are connected through programmable connections to each of the eight OR gates. The diagram uses the array logic convention used in complex circuits. (See Figure 3-22.) Each OR gate must be considered as having 32 inputs. Each output of the decoder is connected through a fuse to one of the inputs of each OR gate. Since each OR gate has 32 internal programmable connections, and since there are eight OR gates, the ROM contains $32 \times 8 = 256$ programmable connections. In general, a $2^k \times n$ ROM will have an internal k -to- 2^k line decoder and n OR gates. Each OR gate has 2^k inputs, which are connected through programmable connections to each of the outputs of the decoder.

Four technologies are used for ROM programming. If mask programming is used, then the ROM is called simply a ROM. If fuses are used, the ROM can be programmed by the user having the proper programming equipment. In this case, the ROM is referred to as a *programmable ROM*, or PROM. If the ROM uses the erasable floating-gate technology, then the ROM is referred to as an *erasable, programmable ROM*, or EPROM. Finally, if the electrically erasable technology is used, the



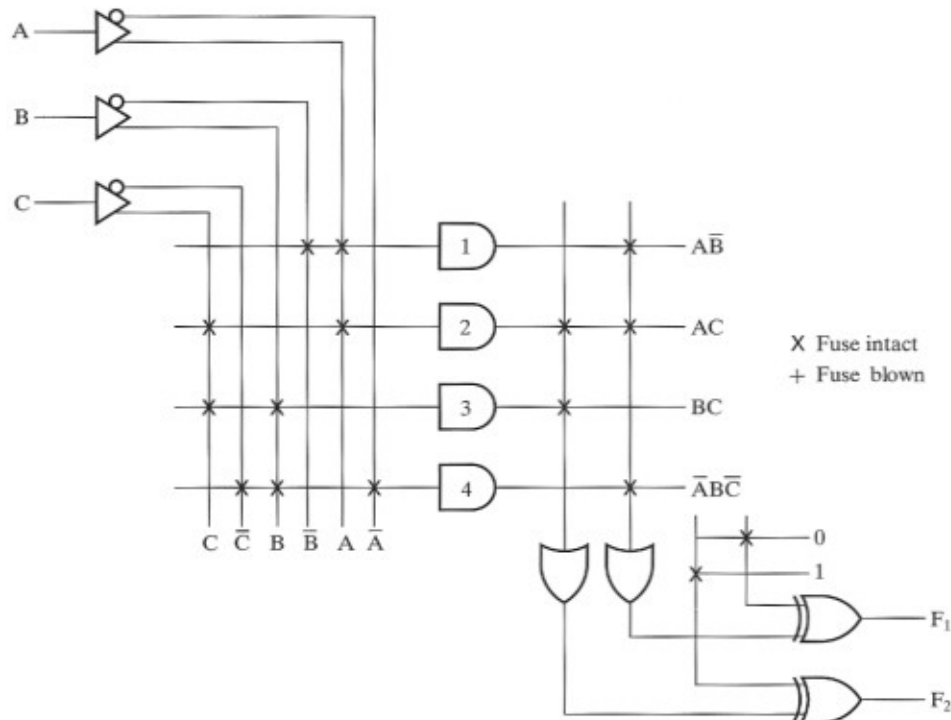
□ **FIGURE 3-25**
Internal Logic of a 32×8 ROM

ROM is referred to an *electrically erasable, programmable ROM*, or EEPROM or E²PROM. As discussed previously, flash memory is a modified version of E²PROM. The choice of programming technology depends on many factors, including the number of identical ROMs to be produced, the desired permanence of the programming, the desire for reprogrammability, and the desired performance in terms of delay.

Programmable Logic Array

The programmable logic array (PLA) is similar in concept to the PROM, except that the PLA does not provide full decoding of the variables and does not generate all the minterms. The decoder is replaced by an array of AND gates that can be programmed to generate product terms of the input variables. The product terms are then selectively connected to OR gates to provide the sum of products for the required Boolean functions.

The internal logic of a PLA with three inputs and two outputs is shown in Figure 3-26. Such a circuit is too small to be cost effective, but is presented here to demonstrate the typical logic configuration of a PLA. The diagram uses the array logic graphics symbols for complex circuits. Each input goes through a buffer and an inverter, represented in the diagram by a composite graphics symbol that has both the true and the complement outputs. Programmable connections run from



□ **FIGURE 3-26**
PLA with Three Inputs, Four Product Terms, and Two Outputs

each input and its complement to the inputs of each AND gate, as indicated by the intersections between the vertical and horizontal lines. The outputs of the AND gates have programmable connections to the inputs of each OR gate. The output of the OR gate goes to an XOR gate, where the other input can be programmed to receive a signal equal to either logic 1 or logic 0. The output is inverted when the XOR input is connected to 1 (since $X \oplus 1 = \bar{X}$). The output does not change when the XOR input is connected to 0 (since $X \oplus 0 = X$). The particular Boolean functions implemented in the PLA of the figure are

$$F_1 = A\bar{B} + AC + \bar{A}B\bar{C}$$

$$F_2 = \overline{AC + BC}$$

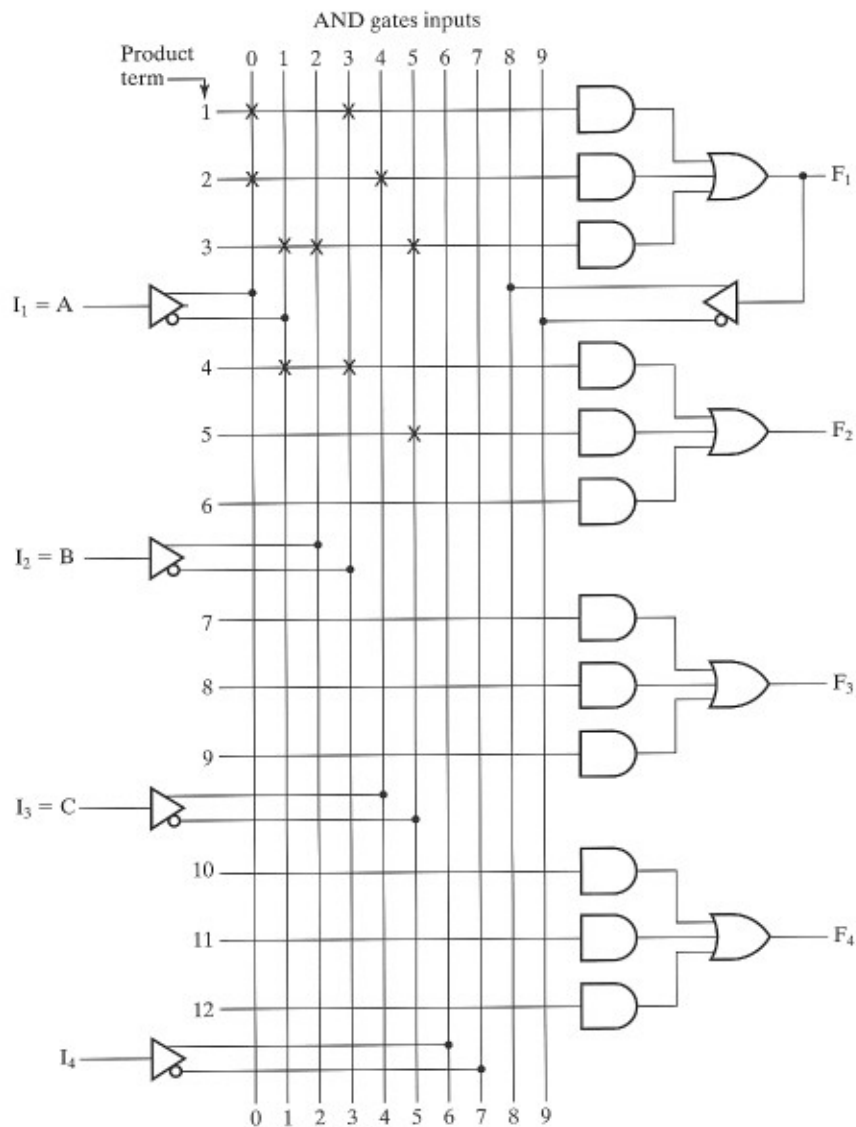
The product terms generated in each AND gate are listed by the output of the gate in the diagram. The product term is determined from the inputs with CLOSED circuit connections. The output of an OR gate gives the logic sum of the selected product terms. The output may be complemented or left in its true form, depending on the programming of the connection associated with the XOR gate.

The size of a PLA is specified by the number of inputs, the number of product terms, and the number of outputs. A typical PLA has 16 inputs, 48 product terms, and eight outputs. For n inputs, k product terms, and m outputs, the internal logic of the PLA consists of n buffer-inverter gates, k AND gates, m OR gates, and m XOR gates. There are $2n \times k$ programmable connections between the inputs and the AND array, $k \times m$ programmable connections between the AND and OR arrays, and m programmable connections associated with the XOR gates.

As with a ROM, the PLA may be mask programmable or field programmable. With mask programming, the customer submits a PLA program table to the manufacturer. The table is used by the vendor to produce a custom-made PLA that has the internal logic specified by the customer. Field programming uses a PLA called a *field-programmable logic array*, or FPLA. This device can be programmed by the user by means of a commercial hardware programming unit.

Programmable Array Logic Devices

The programmable array logic (PAL[®]) device is a PLD with a fixed OR array and a programmable AND array. Because only the AND gates are programmable, the PAL device is easier to program than, but is not as flexible as, the PLA. Figure 3-27 presents the logic configuration of a typical programmable array logic device. The particular device shown has four inputs and four outputs. Each input has a buffer-inverter gate, and each output is generated by a fixed OR gate. The device has four sections, each composed of a three-wide AND-OR array, meaning that there are three programmable AND gates in each section. Each AND gate has 10 programmable input connections, indicated in the diagram by 10 vertical lines intersecting each horizontal line. The horizontal line symbolizes the multiple-input configuration of an AND gate. One of the outputs shown is connected to a buffer-inverter gate and then fed back into the inputs of the AND gates through programmed connections. This is often done with all device outputs.



□ **FIGURE 3-27**
PAL[®] Device with Four Inputs, Four Outputs, and a Three-wide AND-OR Structure

The particular Boolean functions implemented in the PAL in Figure 3-27 are

$$F_1 = A\bar{B} + AC + \bar{A}B\bar{C}$$

$$F_2 = \overline{AC + BC} = \bar{A}\bar{B} + \bar{C}$$

These functions are the same as those implemented using the PLA. Since the output complement is not available, F_2 is converted to a sum of products form.

Commercial PAL devices contain more gates than the one shown in Figure 3-27. A small PAL integrated circuit may have up to eight inputs, eight outputs, and eight sections, each consisting of an eight-wide AND-OR array. Each PAL device output is driven by a three-state buffer and also serves as an input. These input/outputs can be programmed to be an input only, an output only, or bidirectional with a variable signal driving the three-state buffer enable signal. Flip-flops are often included in a PAL device between the array and the three-state buffer at the outputs. Since each output is fed back as an input through a buffer-inverter gate into the AND programmed array, a sequential circuit can be easily implemented.



VLSI PROGRAMMABLE LOGIC DEVICES This supplement, which covers the basics of two typical Field Programmable Gate Arrays (FPGAs) used in course laboratories is available on the Companion Website for the text. The supplement uses multiplexers, adders, flip-flops, latches, and SRAMs. An appendix to the supplement provides a brief introduction to these components.

3-7 CHAPTER SUMMARY

This chapter began with the introduction of two important design concepts, design hierarchy and top-down design, that are used throughout the remainder of the book. Computer-aided design was briefly introduced, with a focus on hardware description languages (HDLs) and logic synthesis.

In Section 3-2, properties of the underlying gate technology were introduced. Two component types with an added output value called high-impedance (Hi-Z), three-state buffers, and transmission gates were described. Important technology parameters, including fan-in, fan-out, and propagation delay, were defined and illustrated. Positive and negative logic conventions describe two different relationships between voltage levels and logical values.

The core of this chapter is a five-step design procedure described in Section 3-3. These steps apply to both manual and computer-aided design. The design begins with a defining specification and proceeds through a formulation step in which the specification is converted to a table or equations. The optimization step applies two-level and multiple-level optimization to obtain a circuit composed of AND gates, OR gates, and inverters. Technology mapping converts this circuit into one that efficiently uses the gates in the available implementation technology. Finally, verification is applied to assure that the final circuit satisfies the initial specification. Three examples illustrate the first three of these steps.

In order to discuss technology mapping, fixed implementation technologies including full custom, standard cell, and gate array approaches were introduced. Cell specification and cell libraries also were introduced, and mapping techniques resembling those used in CAD tools were presented and illustrated for single cell types and multiple cell types.

The final section of the chapter dealt with programmable logic technologies. Three basic technologies—read-only memory, programmable logic arrays, and programmable array logic devices—provide technology mapping alternatives.

REFERENCES

1. HACTEL, G., AND F. SOMENZI. *Logic Synthesis and Verification Algorithms*. Boston: Kluwer Academic Publishers, 1996.
2. DE MICHELI, G. *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, Inc., 1994.
3. KNAPP, S. *Frequently-Asked Questions (FAQ) About Programmable Logic* (<http://www.optimagic.com/faq.html>). OptiMagicTM, Inc., ©1997-2001.
4. LATTICE SEMICONDUCTOR CORPORATION. *Lattice GALs(R)* (<http://www.latticesemi.com/products/spld/GAL/index.cfm>). Lattice Semiconductor Corporation, © 1995-2002.
5. TRIMBERGER, S. M., ED. *Field-Programmable Gate Array Technology*. Boston: Kluwer Academic Publishers, 1994.
6. XILINX, INC., *Xilinx Spartan™-IIE Data Sheet* (http://direct.xilinx.com/bvdocs/publications/ds077_2.pdf). Xilinx, Inc. © 1994-2002.
7. ALTERA(R) CORPORATION, *Altera FLEX 10KE Embedded Programmable Logic Device Family Data Sheet ver. 2.4* (<http://www.altera.com/literature/ds/dsf10ke.pdf>). Altera Corporation, © 1995 - 2002.

PROBLEMS



The plus (+) indicates a more advanced problem and the asterisk (*) indicates a solution is available on the text website.

- 3-1. Design a circuit to implement the following pair of Boolean equations:

$$F = A(C\bar{E} + DE) + \bar{A}D$$

$$F = B(C\bar{E} + DE) + \bar{B}C$$

To simplify drawing the schematic, the circuit is to use a hierarchy based on the factoring shown in the equation. Three instances (copies) of a single hierarchical circuit component made up of two AND gates, an OR gate, and an inverter are to be used. Draw the logic diagram for the hierarchical component and for the overall circuit diagram using a symbol for the hierarchical component.

- 3-2. A hierarchical component with the function

$$H = \bar{X}Y + XZ$$

is to be used along with inverters to implement the following equation:

$$G = \bar{A}\bar{B}C + \bar{A}BD + A\bar{B}\bar{C} + AB\bar{D}$$

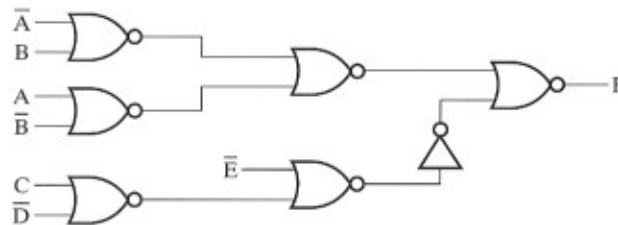
The overall circuit can be obtained by using Shannon's expansion theorem,

$$F = \bar{X} \cdot F_0(X) + X \cdot F_1(X)$$

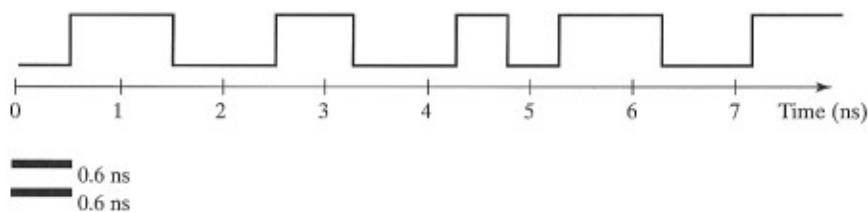
where $F_0(X)$ is F evaluated with variable $X = 0$ and $F_1(X)$ is F evaluated with variable $X = 1$. This expansion F can be implemented with function H by letting $Y = F_0$ and $Z = F_1$. The expansion theorem can then be applied to

each of F_0 and F_1 using a variable in each, preferably one that appears in both true and complemented form. The process can then be repeated until all F_i 's are single literals or constants. For G , use $X = A$ to find G_0 and G_1 and then use $X = B$ for G_0 and G_1 . Draw the top level diagram for G using H as a hierarchical component.

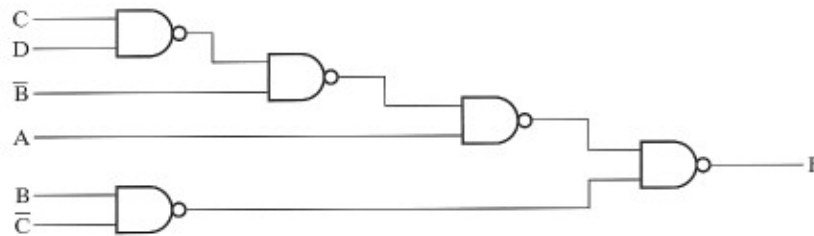
- 3-3. An integrated circuit logic family has NAND gates with a fan-out of 8 standard loads and buffers with a fan-out of 16 standard loads. Sketch a schematic showing how the output signal of a single NAND gate can be applied to 38 other gate inputs using as few buffers as possible. Assume that each input is one standard load.
- 3-4. *The NOR gates in Figure 3-28 have propagation delay $t_{pd} = 0.078$ ns and the inverter has a propagation delay $t_{pd} = 0.052$ ns. What is the propagation delay of the longest path through the circuit?
- 3-5. The waveform in Figure 3-29 is applied to an inverter. Find the output of the inverter, assuming that
- (a) it has no delay.
 - (b) it has a transport delay of 0.06 ns.
 - (c) it has an inertial delay of 0.06 ns with a rejection time of 0.06 ns.
- 3-6. Assume that t_{pd} is the average of t_{PHL} and t_{PLH} . Find the delay from each input to the output in Figure 3-30 by



□ FIGURE 3-28
Circuit for Problem 3-4



□ FIGURE 3-29
Waveform for Problem 3-5



□ **FIGURE 3-30**
Circuit for Problem 3-6

- (a) Finding t_{PHL} and t_{PLH} for each path assuming $t_{\text{PHL}} = 0.30$ ns and $t_{\text{PLH}} = 0.50$ ns for each gate. From these values, find t_{pd} for each path.
 - (b) Using $t_{\text{pd}} = 0.40$ ns for each gate.
 - (c) Compare your answers in part (a) and (b) and discuss any differences.
- 3-7. +The rejection time for inertial delays is required to be less than or equal to the propagation delay. In terms of the discussion of the example in Figure 3-7, why is this condition necessary to determine the delayed output?
 - 3-8. +For a given gate, $t_{\text{PHL}} = 0.05$ ns and $t_{\text{PLH}} = 0.10$ ns. Suppose that an inertial delay model is to be developed from this information for typical gate delay behavior.
 - (a) Assuming a positive output pulse (L H L), what would the propagation delay and rejection time be?
 - (b) Discuss the applicability of the parameters in (a) assuming a negative output pulse (H L H).
 - 3-9. *Show that a positive logic NAND gate is a negative logic NOR gate and vice versa.
 - 3-10. A majority function has an output value of 1 if there are more 1's than 0's on its inputs. The output is 0 otherwise. Design a three-input majority function.
 - 3-11. *Find a function to detect an error in the representation of a decimal digit in BCD. In other words, write an equation with value 1 when the inputs are any one of the six unused bit combinations in the BCD code, and value 0, otherwise.
 - 3-12. Design an Excess-3-to-BCD code converter that gives output code 0000 for all invalid input combinations.
 - 3-13. (a) A low-voltage lighting system is to use a binary logic control for a particular light. This light lies at the intersection point of a T-shaped hallway. There is a switch for this light at each of the three end points of the T. These switches have binary outputs 0 and 1 depending on their position and are

named X_1 , X_2 , and X_3 . The light is controlled by an amplifier driving a thyristor. When Z , the input to the amplifier, is 1, the light is ON and when Z is 0, the light is OFF. You are to find a function $Z = F(X_1, X_2, X_3)$ so that if any one of the switches is changed, the value of Z changes turning the light ON or OFF.

(b) The function for Z is not unique. How many different functions for Z are there?

- 3-14.** +A traffic light control at a simple intersection uses a binary counter to produce the following sequence of combinations on lines A, B, C, and D: 0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, 1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000. After 1000, the sequence repeats, beginning again with 0000, forever. Each combination is present for 5 seconds before the next one appears. These lines drive combinational logic with outputs to lamps RNS (Red - North/South), YNS (Yellow - North/South), GNS (Green - North/South), REW (Red - East/West), YEW (Yellow - East/West), and GEW (Green - East/West). The lamp controlled by each output is ON for a 1 applied and OFF for a 0 applied. For a given direction, assume that green is on for 30 seconds, yellow is on for 5 seconds, and red is on for 45 seconds. (The red intervals overlap for 5 seconds.) Divide up the 80 seconds available for the cycle through the 16 combinations into 16 intervals and determine which lamps should be lit in each interval based on expected driver behavior. Assume that, for interval 0000, a change has just occurred and that GNS = 1, REW = 1, and all other outputs are 0. Design the logic to produce the six outputs using AND and OR gates and inverters.
- 3-15.** Design a combinational circuit that accepts a 3-bit number and generates a 6-bit binary number output equal to the square of the input number.
- 3-16.** +Design a combinational circuit that accepts a 4-bit number and generates a 3-bit binary number output that approximates the square root of the number. For example, if the square root is 3.5 or larger, give a result of 4. If the square root is < 3.5 and ≥ 2.5 , give a result of 3.
- 3-17.** Design a circuit with a 4-bit BCD input A, B, C, D that produces an output W, X, Y, Z that is equal to the input + 6 in binary. For example, 9 (1001) + 6 (0110) = 15 (1111).
- 3-18.** A traffic metering system for controlling the release of traffic from an entrance ramp onto a superhighway has the following specifications for a part of its controller. There are three parallel metering lanes, each with its own stop (red)–go (green) light. One of these lanes, the car pool lane, is given priority for a green light over the other two lanes. Otherwise, a “round robin” scheme in which the green lights alternate is used for the other two (left and right) lanes. The part of the controller that determines which light is to be green (rather than red) is to be designed. The specifications for the controller follow:

Inputs:

- PS - Car pool lane sensor (car present - 1; car absent - 0)
- LS - Left lane sensor (car present - 1; car absent - 0)
- RS - Right lane sensor (car present - 1; car absent - 0)
- RR - Round robin signal (select left - 1; select right - 0)

Outputs:

- PL - Car pool lane light (green - 1; red - 0)
- LL - Left lane light (green - 1; red - 0)
- RL - Right lane light (green - 1; red - 0)

Operation:

1. If there is a car in the car pool lane, PL is 1.
2. If there are no cars in the car pool lane and the right lane, and there is a car in the left lane, LL is 1.
3. If there are no cars in the car pool lane and in the left lane, and there is a car in the right lane, RL is 1.
4. If there is no car in the car pool lane, there are cars in both the left and right lanes, and RR is 1, then LL = 1.
5. If there is no car in the car pool lane, there are cars in both the left and right lanes, and RR is 0, then RL = 1.
6. If any PL, LL, or RL is not specified to be 1 above, then it has value 0.

(a) Find the truth table for the controller part.

(b) Find a minimum multiple-level gate implementation with minimum gate input count using AND gates, OR gates and inverters.

3-19. Complete the design of the BCD-to-seven-segment decoder by performing the following steps:

(a) Plot the seven maps for each of the outputs for the BCD-to-seven-segment decoder specified in Table 3-2.

(b) Simplify the seven output functions in sum-of-products form, and determine the total number of gate inputs that will be needed to implement the decoder.

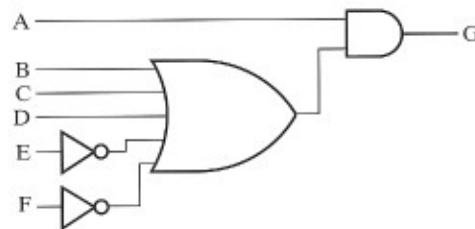
(c) Verify that the seven output functions listed in the text give a valid simplification. Compare the number of gate inputs with that obtained in part (b) and explain the difference.

3-20. +A NAND gate with eight inputs is required. For each of the following cases, minimize the number of gates used in the multiple-level result:

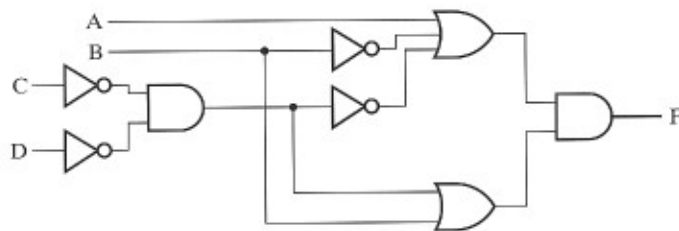
(a) Design the 8-input NAND gate using 2-input NAND gates and NOT gates.

(b) Design the 8-input NAND gate using 2-input NAND gates, 2-input NOR gates, and NOT gates only if needed.

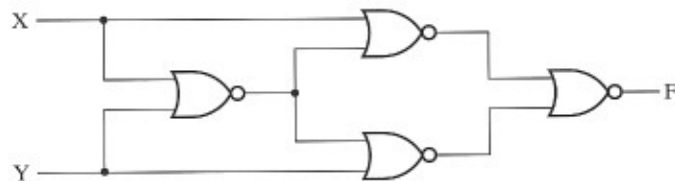
(c) Compare the number of gates used in (a) and (b).



□ **FIGURE 3-31**
Circuit for Problem 3-21



□ **FIGURE 3-32**
Circuit for Problem 3-22

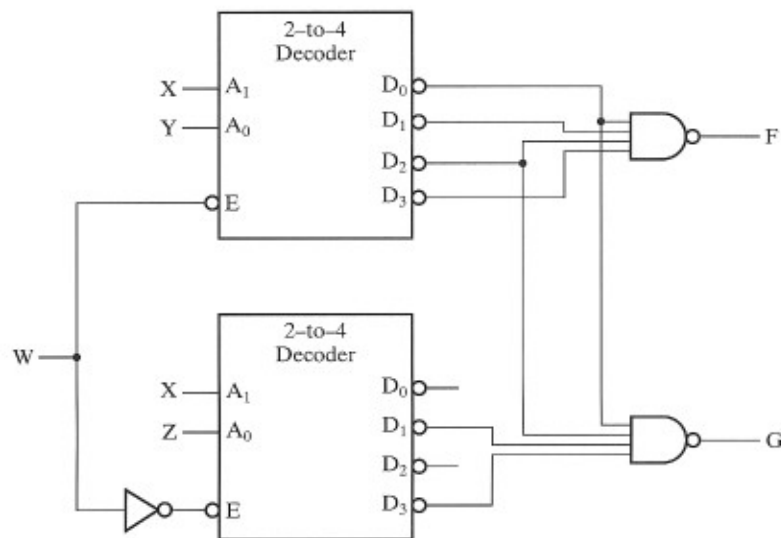


□ **FIGURE 3-33**
Circuit for Problem 3-23

- 3-21.** Perform a minimum-cost (use minimum total normalized area as cost) technology mapping using NAND cells and inverters from Table 3-3 for the circuit shown in Figure 3-31.
- 3-22.** Perform a low-cost (use minimum total normalized area as cost) technology mapping using all cells from Table 3-3 for the circuit shown in Figure 3-32.
- 3-23.** By using manual methods, verify that the circuit of Figure 3-33 generates the exclusive-NOR function.
- 3-24.** *Manually verify that the functions for the outputs F and G of the hierarchical circuit in Figure 3-34 are

$$F = \bar{X}Y + X\bar{Y}Z + XY\bar{Z}$$

$$G = XZ + \bar{X}YZ + \bar{X}\bar{Y}\bar{Z}$$

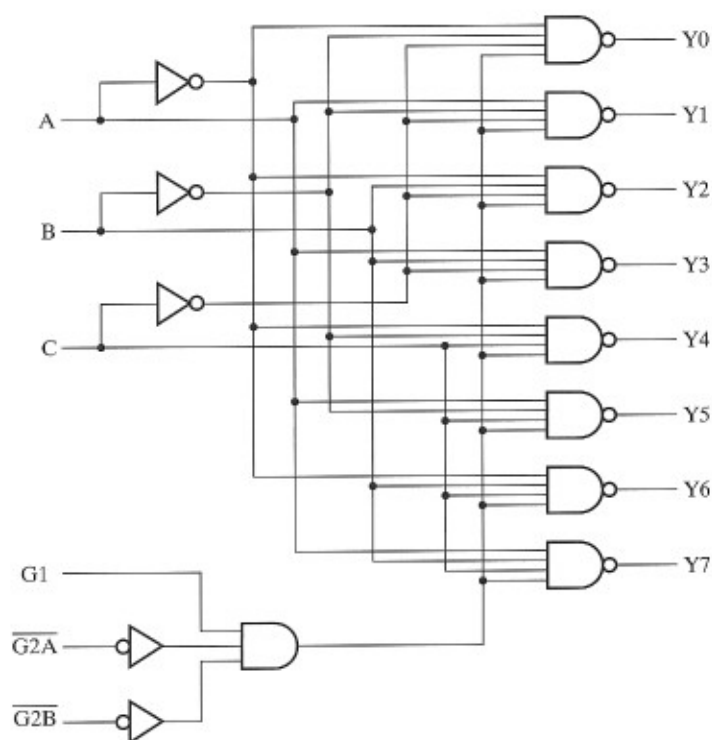


□ **FIGURE 3-34**
Circuit for Problems 3-24 and 3-25

3-25. Manually verify that the truth table for the outputs F and G of the hierarchical circuit in Figure 3-34 is as follows:

W	X	Y	Z	F	G
0	0	0	0	1	1
0	0	0	1	1	1
0	0	1	0	1	0
0	0	1	1	1	0
0	1	0	0	1	1
0	1	0	1	1	1
0	1	1	0	1	0
0	1	1	1	1	0
1	0	0	0	0	0
1	0	0	1	0	1
1	0	1	0	0	0
1	0	1	1	0	1
1	1	0	0	0	0
1	1	0	1	0	1
1	1	1	0	0	0
1	1	1	1	0	1

3-26. The logic diagram for a 74HC138 MSI CMOS circuit is given in Figure 3-35. Find the Boolean function for each of the outputs. Describe the circuit function carefully.



□ **FIGURE 3-35**
Circuit for Problem 3-26 and Problem 3-27

- 3-27.** Do Problem 3-26 by using logic simulation to find the output waveforms of the circuit or a partial truth table listing, rather than finding Boolean functions.
- 3-28.** In Figure 3-21, simulation results are given for the BCD-to-Excess-3 Code Converter for the BCD inputs for 0 through 9. Perform a similar logic simulation to determine the results for BCD inputs 10 through 15.

CHAPTER

4

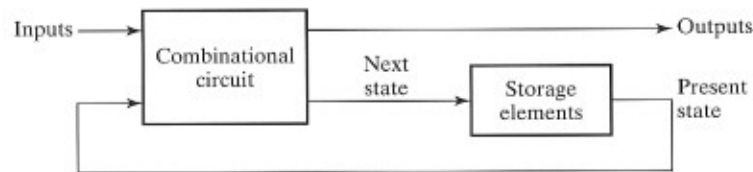
COMBINATIONAL FUNCTIONS AND CIRCUITS

In this chapter, we will learn about a number of functions and the corresponding fundamental circuits that are very useful in designing larger digital circuits. The fundamental, reusable circuits, which we call functional blocks, implement functions of a single variable, decoders, encoders, code converters, multiplexers, and programmable logic. Aside from being important building blocks for larger circuits and systems, many of these functions are strongly tied to the various components of hardware description languages and serve as a vehicle for HDL presentation. As an alternative to truth tables, equations, and schematics, VHDL and Verilog hardware description languages are introduced.

In the generic computer diagram at the beginning of Chapter 1, multiplexers are very important for selecting data in the processor, in memory, and on I/O boards. Decoders are used for selecting boards attached to the input-output bus and to decode instructions to determine the operations performed in the processor. Encoders are used in a number of components, such as the keyboard. Programmable logic is used to handle complex instructions within processors as well as in many other components of the computer. Overall, functional blocks are widely used, so concepts from this chapter apply across most components of the generic computer, including memories.

4-1 COMBINATIONAL CIRCUITS

In Chapter 3, we defined and illustrated combinational circuits and their design. In this section, we define specific combinational functions and corresponding



□ **FIGURE 4-1**
Block Diagram of a Sequential Circuit

combinational circuits, referred to as *functional blocks*. In some cases, we will go through the design process for obtaining a circuit from the function, while in other cases, we will simply present the function and an implementation of it. These functions have special importance in digital design. In the past, the functional blocks were manufactured as small and medium scale integrated circuits. Today, in very large scale integrated (VLSI) circuits, functional blocks are used to design circuits with many such blocks. Combinational functions and their implementations are fundamental to the understanding of VLSI circuits. By using a hierarchy, we typically construct circuits as instances of these functions or the associated functional blocks.

Large-scale and very large-scale integrated circuits are almost always sequential circuits as described in Section 3.1 and detailed beginning in Chapter 6. The functions and functional blocks discussed in this chapter, are combinational. However, they are often combined with storage elements to form sequential circuits as shown in Figure 4-1. Inputs to the combinational circuit can come from both the external environment and from the storage elements. Outputs from the combinational circuit go to both the external environment and to the storage elements. In later chapters, we use the combinational functions and blocks defined here, in Chapter 5, and in Chapter 6 with storage elements to form sequential circuits that perform very useful functions. Further, the functions and blocks defined here and in Chapter 5 serve as a basis for describing and understanding both combinational and sequential circuits using hardware description languages in this and subsequent chapters.

4-2 RUDIMENTARY LOGIC FUNCTIONS

Value-fixing, transferring, inverting, and enabling are among the most elementary of combinational logic functions. The first two operations, value-fixing and transferring, do not involve any Boolean operators. They use only variables and constants. As a consequence, logic gates are not involved in the implementation of these operations. Inverting involves only one logic gate per variable, and enabling involves one or two logic gates per variable.

Value-Fixing, Transferring, and Inverting

If a single-bit function depends on a single variable X , at most, four different functions are possible. Table 4-1 gives the truth tables for these functions. The first and last columns of the table assign either constant value 0 or constant value 1 to the

□ **TABLE 4-1**
Functions of One Variable

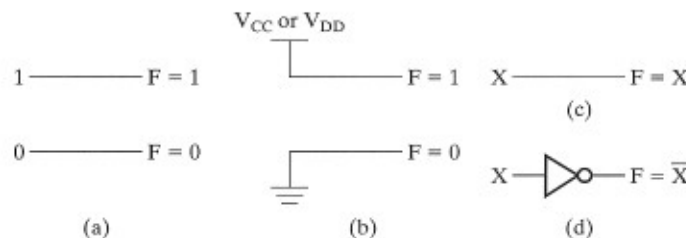
X	$F = 0$	$F = X$	$F = \bar{X}$	$F = 1$
0	0	0	1	1
1	0	1	0	1

function, thus performing *value-fixing*. In the second column, the function is simply the input variable X , thus *transferring* X from input to output. In the third column, the function is \bar{X} , thus *inverting* input X to become output \bar{X} .

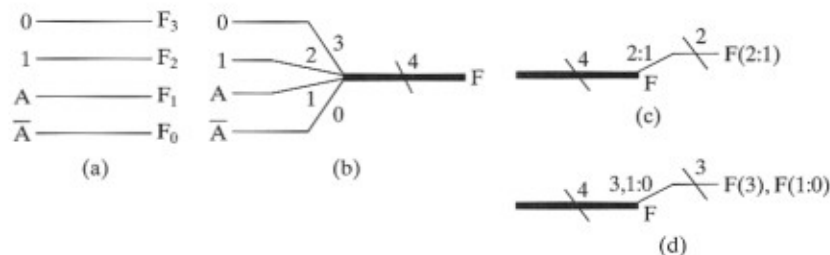
The implementations for these four functions are given in Figure 4-2. Value fixing is implemented by connecting a constant 0 or constant 1 to output F as shown in Figure 4-2(a). Figure 4-2(b) shows alternative representations used in logic schematics. For the positive logic convention, constant 0 is represented by the electrical ground symbol and constant 1 by a power supply voltage symbol. The latter symbol is labeled with either V_{CC} or V_{DD} . Transferring is implemented by a simple wire connecting X to F as in Figure 4-2(c). Finally, inverting is represented by an inverter which forms $F = \bar{X}$ from input X as shown in Figure 4-2(d).

Multiple-Bit Functions

The functions defined so far can be applied to multiple bits on a bitwise basis. We can think of these multiple-bit functions as vectors of single bit functions. For example, suppose that we have four functions, F_3 , F_2 , F_1 , and F_0 that make up a four-bit function F . We can order the four functions with F_3 as the most significant bit and F_0 the least significant bit, providing the vector $F = (F_3, F_2, F_1, F_0)$. Suppose that F consists of rudimentary functions $F_3 = 0$, $F_2 = 1$, $F_1 = A$ and $F_0 = \bar{A}$. Then we can write F as the vector $(0, 1, A, \bar{A})$. For $A = 0$, $F = (0, 1, 0, 1)$ and for $A = 1$, $F = (0, 1, 1, 0)$. This multiple-bit function can be referred to as $F(3:0)$ or simply as F and is implemented in Figure 4-3(a). For convenience in schematics, we often represent a set of multiple, related wires by using a single line of greater thickness with a slash across the line. An integer giving the number of wires represented accompanies the slash as shown in Figure 4-3(b). In order to connect the values 0, 1, X , and \bar{X} to the



□ **FIGURE 4-2**
 Implementation of Functions of a Single Variable X



□ **FIGURE 4-3**
Implementation of Multi-bit Rudimentary Functions

appropriate bits of F , we break F up into four wires with each wire labeled with the bit of F . Also, in the process of transferring, we may wish to use only a subset of the elements in F , for example, F_2 and F_1 . The notation for the bits of F can be used for this purpose as shown in Figure 4-3(c). In Figure 4-3(d), a more complex case illustrates the use of F_3 , F_1 , F_0 at a destination. Note that since F_3 , F_1 , and F_0 are not all together, we cannot use the range notation $F(3:0)$ to denote this subvector. Instead, we must use a combination of two subvectors, $F(3)$, $F(1:0)$ denoted by subscripts 3, 1:0. The actual notation used for vectors and subvectors varies among the schematic drawing tools or HDL tools available. Figure 4-3 illustrates just one approach. For a specific tool, the documentation should be consulted.

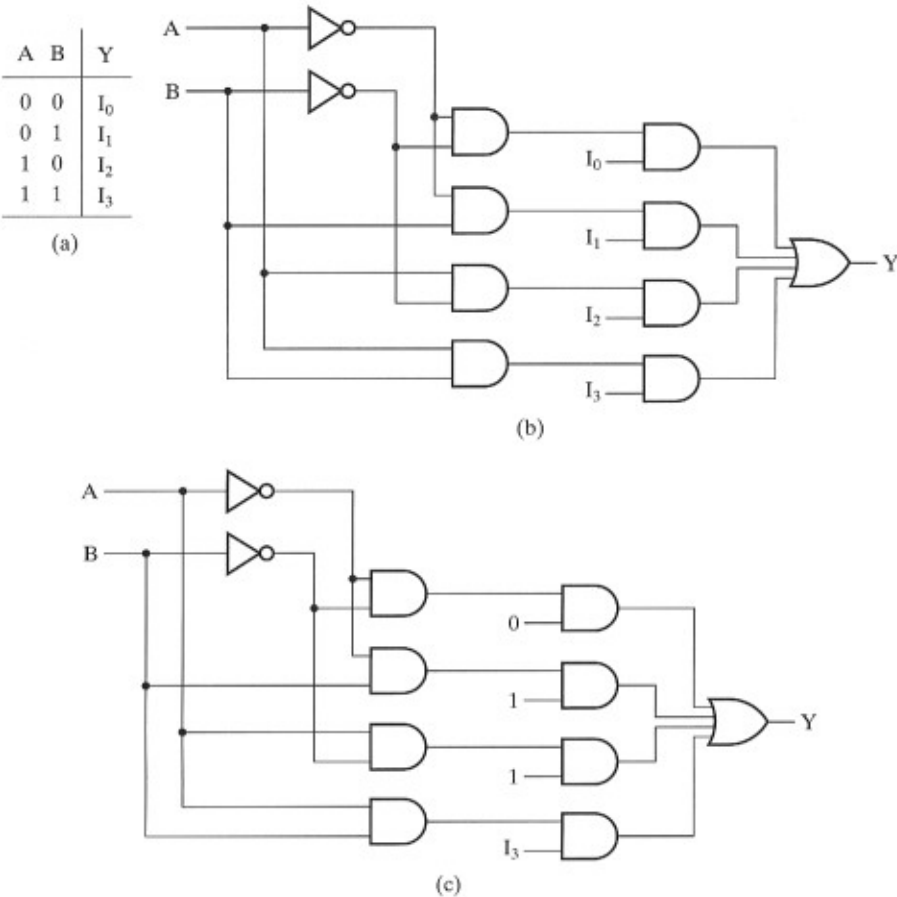
Value-fixing, transferring and inverting have a variety of applications in logic design. Value-fixing involves replacing one or more variables with constant values 1 and 0. Value fixing may be permanent or temporary. In permanent value-fixing, the value can never be changed. In temporary value-fixing, the values can be changed, often by mechanisms that are somewhat different than those employed in ordinary logical operation. A major application of fixed and temporary value-fixing is in programmable logic devices. Any logic function that is within the capacity of the programmable device can be implemented by fixing a set of values, as illustrated in the next example.

EXAMPLE 4-1 Value-Fixing for Implementing a Function

Consider the truth table shown in Figure 4-4(a). A and B are the input variables and I_0 through I_3 are also variables. Values 0 and 1 can be assigned to I_0 through I_3 depending upon the desired function. Note that Y is actually a function of six variables giving a fully-expanded truth table containing 64 rows and seven columns. But, by putting I_0 through I_3 in the output column, we considerably reduce the size of the table. The equation for the output Y for this truth table is

$$Y(A, B, I_0, I_1, I_2, I_3) = \bar{A} \bar{B} I_0 + \bar{A} B I_1 + A \bar{B} I_2 + A B I_3$$

The implementation of this equation is given in Figure 4-4(b). By fixing the values of I_0 through I_3 , we can implement any function $Y(A, B)$. As shown in Table 4-2, we can implement $Y = A + B$ by using $I_0 = 0$, $I_1 = 1$, $I_2 = 1$, and $I_3 = 1$. Or, we can implement $Y = A \bar{B} + \bar{A} B$ by using $I_0 = 0$, $I_1 = 1$, $I_2 = 1$, and $I_3 = 0$. Either of these functions can



□ **FIGURE 4-4**
Implementation of Two Functions by Using Value-Fixing

□ **TABLE 4-2**
Function Implementation by Value-Fixing

A	B	$Y = A + B$	$Y = A\bar{B} + \bar{A}B$	$Y = A + B$ ($I_3 = 1$) or $Y = A\bar{B} + \bar{A}B$ ($I_3 = 0$)
0	0	0	0	0
0	1	1	1	1
1	0	1	1	1
1	1	1	0	I_3

be implemented permanently, or can be implemented temporarily by fixing $I_0 = 0$, $I_1 = 1$, $I_2 = 1$, and using I_3 as a variable with $I_3 = 1$ for $A + B$ and $I_3 = 0$ for $A\bar{B} + \bar{A}B$. The final circuit is shown in Figure 4-4(c). ■

Enabling

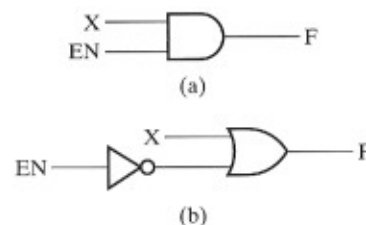
The concept of enabling a signal first appeared in Section 2-9 where Hi-Z outputs and three-state buffers were introduced. In general, enabling permits an input signal to pass through to an output. In addition to replacing the input signal with the Hi-Z state, disabling also can replace the input signal with a fixed output value, either 0 or 1. The additional input signal, often called *ENABLE* or *EN*, is required to determine whether the output is enabled or not. For example, if *EN* is 1, then the input *X* reaches the output (enabled) but if *EN* is 0, the output is fixed at 0 (disabled). For this case, with the disabled value at 0, the input signal is ANDed with the *EN* signal as shown in Figure 4-5(a). If the disabled value is 1, then the input signal *X* is ORed with the complement of the *EN* signal as shown in Figure 4-5(b). Alternatively, the signal to the output may be enabled with $EN = 0$ instead of 1 and the *EN* signal denoted as \overline{EN} as *EN* is inverted as illustrated in Figure 4-5(b).

EXAMPLE 4-2 Enabling Application

In most automobiles, the lights, radio, and power windows operate only if the ignition switch is turned on. In this case, the ignition switch acts as an “enabling” signal. Suppose that we model this automotive subsystem using the following variables and definitions:

- Ignition switch *IG* - Value 0 if off and value 1 if on
- Light Switch *LS* - Value 0 if off and value 1 if on
- Radio Switch *RS* - Value 0 if off and value 1 if on
- Power Window Switch *WS* - Value 0 if off and value 1 if on
- Lights *L* - Value 0 if off and value 1 if on
- Radio *R* - Value 0 if off and value 1 if on
- Power Windows *W* - Value 0 if off and value 1 if on

Table 4-3 contains the condensed truth table for the operation of this automobile subsystem. Note that when the ignition switch *IS* is off (0), all of the controlled accessories are off (0) regardless of their switch settings. This is indicated by the first row of the table. With the use of X's, this condensed truth table with just nine



□ **FIGURE 4-5**
Enabling Circuits

□ TABLE 4-3
Truth Table for Enabling Application

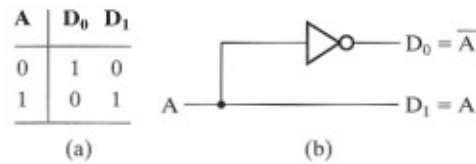
Input Switches				Accessory Control		
IS	LS	RS	WS	L	R	W
0	X	X	X	0	0	0
1	0	0	0	0	0	0
1	0	0	1	0	0	1
1	0	1	0	0	1	0
1	0	1	1	0	1	1
1	1	0	0	1	0	0
1	1	0	1	1	0	1
1	1	1	0	1	1	0
1	1	1	1	1	1	1

rows represents the same information as the usual 16-row truth table. Whereas X's in output columns represent don't-care conditions, X's in input columns are used to represent product terms that are not minterms. For example, 0XXX represents the product term \overline{IS} . Just as with minterms, each variable is complemented if the corresponding bit in the input combination from the table is 0 and is not complemented if the bit is 1. If the corresponding bit in the input combination is an X, then the variable does not appear in the product term. When the ignition switch *IS* is on (1), then the accessories are controlled by their respective switches. When *IS* is off (0), all accessories are off. So *IS* replaces the normal values of the outputs *L*, *R*, and *W* with a fixed value 0 and meets the definition of an *ENABLE* signal. ■

4-3 DECODING

In digital computers, discrete quantities of information are represented by binary codes. An n -bit binary code is capable of representing up to 2^n distinct elements of coded information. Decoding is the conversion of an n -bit input code to an m -bit output code with $n \leq m \leq 2^n$ such that each valid input code word produces a unique output code. Decoding is performed by a *decoder*, a combinational circuit with an n -bit binary code applied to its inputs and an m -bit binary code appearing at the outputs. The decoder may have unused bit combinations on its inputs for which no corresponding m -bit code appears at the outputs. Among all of the specialized functions defined here, decoding is the most important since this function and the corresponding functional blocks are incorporated into many of the other functions and functional blocks defined here.

In this section, the functional blocks that implement decoding are called n -to- m -line decoders, where $m \leq 2^n$. Their purpose is to generate the 2^n (or fewer) minterms from the n input variables. For $n = 1$ and $m = 2$, we obtain the 1-to-2-line decoding function with input *A* and outputs D_0 and D_1 . The truth table for this



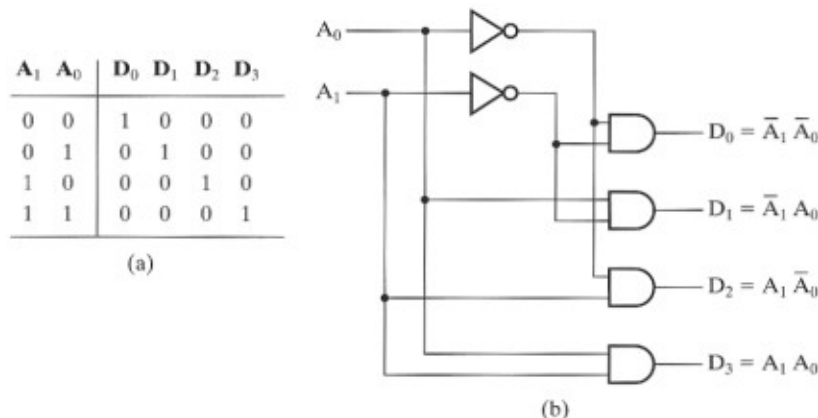
□ **FIGURE 4-6**
A 1-to-2-Line Decoder

decoding function is given in Figure 4-6(a). If $A = 0$, then $D_0 = 1$ and $D_1 = 0$. If $A = 1$, then $D_0 = 0$ and $D_1 = 1$. From this truth table, $D_0 = \bar{A}$ and $D_1 = A$ giving the circuit shown in Figure 4-6(b).

A second decoding function for $n = 2$ and $m = 4$ with the truth table given in Figure 4-7(a) better illustrates the general nature of decoders. This table has 2-variable minterms as its outputs, with each row containing one output value equal to 1 and three output values equal to 0. Output D_i is equal to 1 whenever the two input values on A_1 and A_0 are the binary code for the number i . As a consequence, the circuit implements the four possible minterms of two variables, one minterm for each output. In the logic diagram in Figure 4-7(b), each minterm is implemented by a 2-input AND gate. These AND gates are connected to two 1-to-2-line decoders, one for each of the lines driving the AND gate inputs.

Decoder Expansion

Large decoders can be constructed by simply implementing each minterm function using a single AND gate with more inputs. Unfortunately, as decoders become



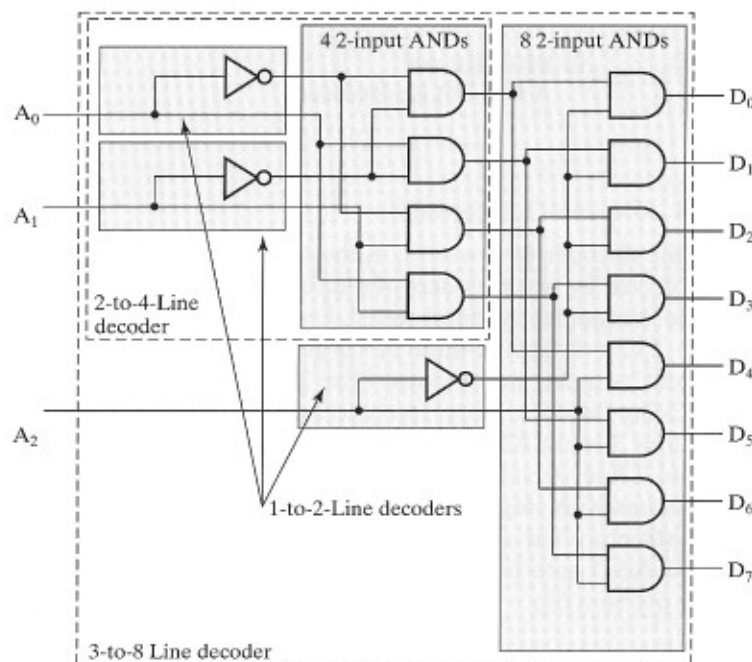
□ **FIGURE 4-7**
A 2-to-4-Line Decoder

larger, this approach gives a high gate input count. In this section, we give a procedure that uses design hierarchy and collections of AND gates to construct any decoder with n inputs and 2^n outputs. The resulting decoder has the same or a lower gate input count than the one constructed by simply enlarging each AND gate.

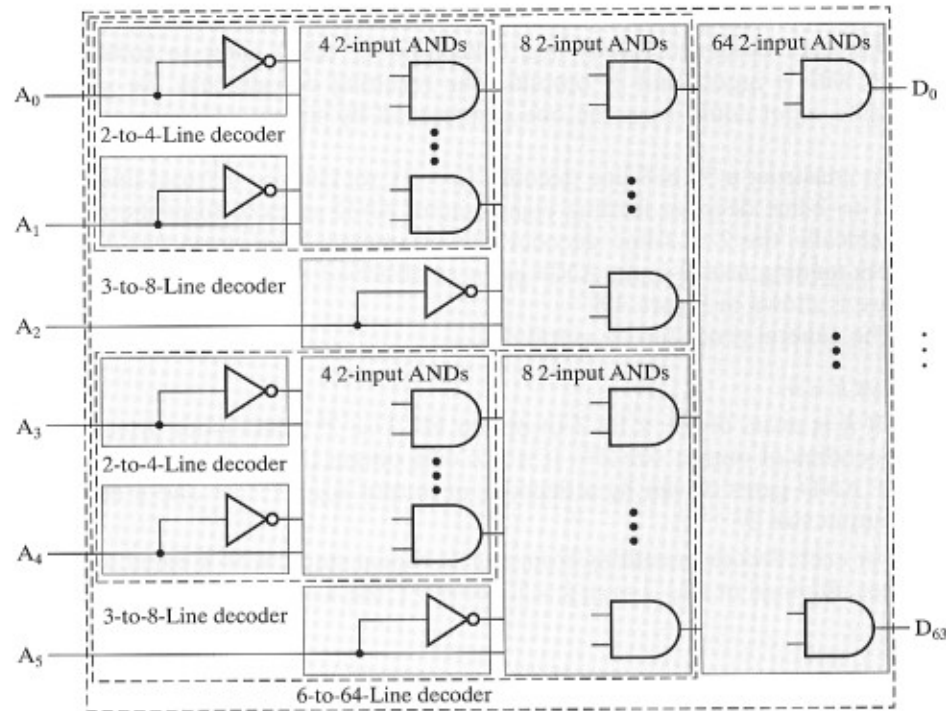
To construct a 3-to-8-line decoder ($n = 3$), we can use a 2-to-4-line decoder and a 1-to-2-line decoder feeding eight 2-input AND gates to form the minterms. Hierarchically, the 2-to-4-line decoder can be implemented using two 1-to-2-line decoders feeding four 2-input AND gates as observed in Figure 4-7. The resulting structure is shown in Figure 4-8.

The general procedure is as follows:

1. Let $k = n$.
2. If k is even, divide k by 2 to obtain $k/2$. Use 2^k AND gates driven by two decoders of output size $2^{k/2}$. If k is odd, obtain $(k + 1)/2$ and $(k - 1)/2$. Use 2^k AND gates driven by a decoder of output size $2^{(k + 1)/2}$ and a decoder of output size $2^{(k - 1)/2}$.
3. For each decoder resulting from step 2, repeat step 2 with k equal to the values obtained in step 2 until $k = 1$. For $k = 1$, use a 1-to-2 decoder.



□ FIGURE 4-8
A 3-to-8-Line Decoder



□ **FIGURE 4-9**
A 6-to-64-Line Decoder

EXAMPLE 4-3 6-to-64-Line Decoder

For a 6-to-64-line decoder ($k = n = 6$), in the first execution of step 2, 64 2-input AND gates are driven by two decoders of output size $2^3 = 8$ (i.e., by two 3-to-8-line decoders). In the second execution of step 2, $k = 3$. Since k is odd, the result is $(k + 1)/2 = 2$ and $(k - 1)/2 = 1$. Eight 2-input AND gates are driven by a decoder of output size $2^2 = 4$ and by a decoder of output size $2^1 = 2$ (i.e., by a 2-to-4-line decoder and by a 1-to-2-line decoder, respectively). Finally, on the next execution of step 2, $k = 2$, giving four 2-input AND gates driven by two decoders with output size 2 (i.e., by two 1-to-2-line decoders). Since all decoders have been expanded, the algorithm terminates with step 3 at this point. The resulting structure is shown in Figure 4-9. This structure has a gate input count of $6 + 2(2 \times 4) + 2(2 \times 8) + 2 \times 64 = 182$. If a single AND gate for each minterm were used, the resulting gate input count would be $6 + (6 \times 64) = 390$, so a substantial gate input count reduction has been achieved. ■

As an alternative expansion situation, suppose that multiple decoders are needed and that the decoders have common input variables. In this case, instead of

implementing separate decoders, parts of the decoders can be shared. For example, suppose that three decoders d_a , d_b , and d_c are functions of input variables as follows:

$$\begin{aligned}d_a & (A, B, C, D) \\d_b & (A, B, C, E) \\d_c & (C, D, E, F)\end{aligned}$$

A 3-to-8-line decoder for A , B , and C can be shared between d_a and d_b . A 2-to-4-line decoder for C and D can be shared between d_a and d_c . A 2-to-4-line decoder for C and E can be shared between d_b and d_c . If we implement all of this sharing, we would have C entering three different decoders and the circuit would be redundant. To use C just once in shared decoders larger than 1 to 2, we can consider the following distinct cases:

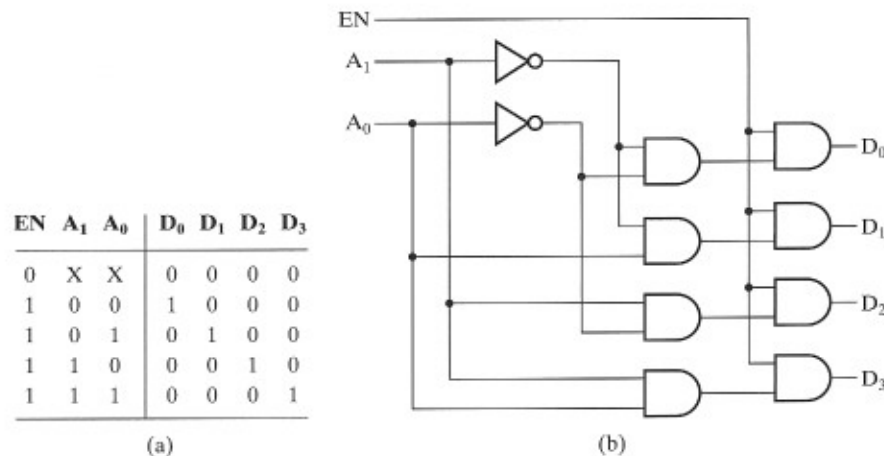
1. (A, B) shared by d_a and d_b , and (C, D) shared by d_a and d_c
2. (A, B) shared by d_a and d_b , and (C, E) shared by d_b and d_c , or
3. (A, B, C) shared by d_a and d_b .

Since cases 1 and 2 will clearly have the same costs, we will compare the cost of cases 1 and 3. For case 1, the costs of functions d_a , d_b , and d_c will be reduced by the cost of two 2-to-4-line decoders (exclusive of inverters) or 16 gate inputs. For case 3, the costs for functions d_a and d_b are reduced by one 3-to-8-line decoder (exclusive of inverters) or 24 gate inputs. So case 3 should be implemented. Formalization of this procedure into an algorithm is beyond our current scope, so only this illustration of the approach is given.

Decoder and Enabling Combinations

The function, n -to- m -line decoding with enabling, can be implemented by attaching m enabling circuits to the decoder outputs. Then, m copies of the enabling signal EN are attached to the enable control inputs of the enabling circuits. For $n = 2$ and $m = 4$, the resulting 2-to-4-line decoder with enable is shown in Figure 4-10, along with its truth table. For $EN = 0$, all of the outputs of the decoder are 0. For $EN = 1$, one of the outputs of the decoder, determined by the value on (A_1, A_0) , is 1 and all others are 0. If the decoder is controlling a set of lights, then with $EN = 0$, all lights are off, and with $EN = 1$, exactly one light is on, with the other three off. For large decoders ($n \geq 4$), the gate input count can be reduced by placing the enable circuits on the inputs to the decoder and their complements rather than on each of the decoder outputs.

In Section 4-5, selection using multiplexers will be covered. The inverse of selection is *distribution* in which information received from a single line is transmitted to one of 2^n possible output lines. The circuit which implements such distribution is called a *demultiplexer*. The specific output to which the input signal is transmitted is controlled by the bit combination on n selection lines. The 2-to-4-line decoder with enable in Figure 4-10 is an implementation of a 1-to-4-line demultiplexer. For the demultiplexer, input EN provides the data, while the other



□ **FIGURE 4-10**
A 2-to-4-Line Decoder with Enable

inputs act as the selection variables. Although the two circuits have different applications, their logic diagrams are exactly the same. For this reason, a decoder with enable input is referred to as a decoder/demultiplexer. The data input EN has a path to all four outputs, but the input information is directed to only one of the outputs, as specified by the two selection lines A_1 and A_0 . For example, if $(A_1, A_0) = 10$, output D_2 has the value applied to input EN , while all other outputs remain inactive at logic 0. If the decoder is controlling a set of four lights, with $(A_1, A_0) = 10$ and EN periodically changing between 1 and 0, the light controlled by D_2 flashes on and off and all other lights are off.

4-4 ENCODING

An encoder is a digital function that performs the inverse operation of a decoder. An encoder has 2^n (or fewer) input lines and n output lines. The output lines generate the binary code corresponding to the input value. An example of an encoder is the octal-to-binary encoder whose truth table is given in Table 4-4. This encoder has eight inputs, one for each of the octal digits, and three outputs that generate the corresponding binary number. It is assumed that only one input has a value of 1 at any given time, so that the table has only eight rows with specified output values. For the remaining 56 rows, all of the outputs are don't cares. From the truth table, we can observe that A_i is 1 for the columns in which D_j is 1 only if subscript j has a binary representation with a 1 in the i th position. For example, output $A_0 = 1$ if the input is 1 or 3 or 5 or 7. Since all of these values are odd, they have a 1 in the 0 position of their binary representation. This approach can be used to find the truth table. From the table, the encoder can be implemented with n OR gates, one for each output variable A_i . Each OR gate

□ **TABLE 4-4**
Truth Table for Octal-to-Binary Encoder

Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

combines the input variables D_j having a 1 in the rows for which A_i has value 1. For the 8-to-3-line encoder, the resulting output equations are

$$A_0 = D_1 + D_3 + D_5 + D_7$$

$$A_1 = D_2 + D_3 + D_6 + D_7$$

$$A_2 = D_4 + D_5 + D_6 + D_7$$

which can be implemented with three 4-input OR gates.

The encoder just defined has the limitation that only one input can be active at any given time: if two inputs are active simultaneously, the output produces an incorrect combination. For example, if D_3 and D_6 are 1 simultaneously, the output of the encoder will be 111 because all the three outputs are equal to 1. This represents neither a binary 3 nor a binary 6. To resolve this ambiguity, some encoder circuits must establish an input priority to ensure that only one input is encoded. If we establish a higher priority for inputs with higher subscript numbers, and if both D_3 and D_6 are 1 at the same time, the output will be 110 because D_6 has higher priority than D_3 . Another ambiguity in the octal-to-binary encoder is that an output of all 0's is generated when all the inputs are 0, but this output is the same as when D_0 is equal to 1. This discrepancy can be resolved by providing a separate output to indicate that at least one input is equal to 1.

Priority Encoder

A priority encoder is a combinational circuit that implements a priority function. As mentioned in the preceding paragraph, the operation of the priority encoder is such that if two or more inputs are equal to 1 at the same time, the input having the highest priority takes precedence. The truth table for a four-input priority encoder is given in Table 4-5. With the use of X's, this condensed truth table with just five rows represents the same information as the usual 16-row truth table. Whereas X's in output columns represent don't-care conditions, X's in input

□ **TABLE 4-5**
Truth Table of Priority Encoder

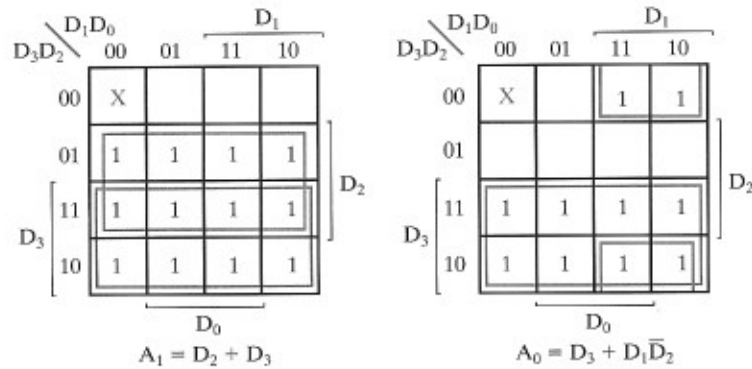
Inputs				Outputs		
D_3	D_2	D_1	D_0	A_1	A_0	V
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1

columns are used to represent product terms that are not minterms. For example, 001X represents the product term $\overline{D}_3 \overline{D}_2 D_1$. Just as with minterms, each variable is complemented if the corresponding bit in the input combination from the table is 0 and is not complemented if the bit is 1. If the corresponding bit in the input combination is an X, then the variable does not appear in the product term. Thus, for 001X, the variable D_0 , corresponding to the position of the X, does not appear in $\overline{D}_3 \overline{D}_2 D_1$.

The number of rows of a full truth table represented by a row in the condensed table is 2^p , where p is the number of X's in the row. For example, in Table 4-5, 1XXX represents $2^3 = 8$ truth table rows, all having the same value for all outputs. In forming a condensed truth table, we must include each minterm in at least one of the rows in the sense that minterm can be obtained by filling in 1's and 0's for the X's. Also, a minterm must never be included in more than one row such that the rows in which it appears have one or more conflicting output values.

We form Table 4-5 as follows: Input D_3 has the highest priority; so, regardless of the values of the other inputs, when this input is 1, the output for $A_1 A_0$ is 11 (binary 3). From this we obtain the last row of the table. D_2 has the next priority level. The output is 10 if $D_2 = 1$, provided that $D_3 = 0$, regardless of the values of the lower priority inputs. From this, we obtain the fourth row of the table. The output for D_1 is generated only if all inputs with higher priority are 0, and so on down the priority levels. From this, we obtain the remaining rows of the table. The valid output designated by V is set to 1 only when one or more of the inputs are equal to 1. If all inputs are 0, V is equal to 0, and the other two outputs of the circuit are not used and are specified as don't-care conditions in the output part of the table.

The maps for simplifying outputs A_1 and A_0 are shown in Figure 4-11. The minterms for the two functions are derived from Table 4-5. The output values in the table can be transferred directly to the maps by placing them in the squares covered by the corresponding product term represented in the table. The optimized equation for each function is listed under the map for the function. The



□ **FIGURE 4-11**
Maps for Priority Encoder

equation for output V is an OR function of all the input variables. The priority encoder is implemented in Figure 4-12 according to the following Boolean functions:

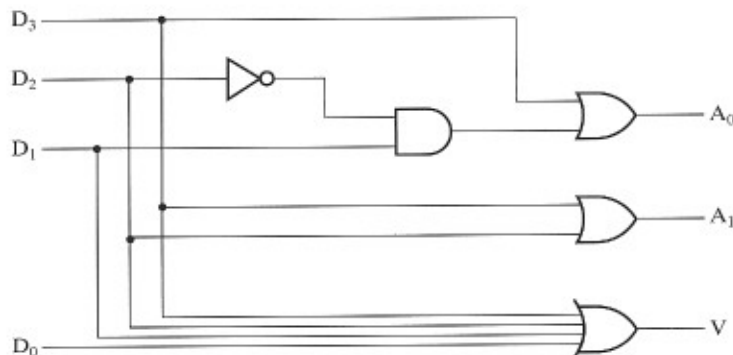
$$A_0 = D_3 + D_1 \bar{D}_2$$

$$A_1 = D_2 + D_3$$

$$V = D_0 + D_1 + D_2 + D_3$$

Encoder Expansion

Thus far, we have considered only small encoders. Encoders can be expanded to larger numbers of inputs by expanding OR gates. In the implementation of



□ **FIGURE 4-12**
Logic Diagram of a 4-Input Priority Encoder

decoders, the use of multiple-level circuits with OR gates beyond the output levels shared in implementing the more significant bits in the output codes reduces the gate input count as n increases for $n \geq 5$. For $n \geq 3$, multiple-level circuits result from technology mapping anyway due to limited gate fan-in. Designing multiple-level circuits with shared gates reduces the cost of the encoders after technology mapping.

4-5 SELECTING

Selection of information to be used in a computer is a very important function, not only in communicating between the parts of the system, but also within the parts as well. Among other uses, selecting in combination with value fixing can implement combinational functions. Circuits that perform selection typically have a set of inputs from which selections are made, a single output, and a set of control lines for making the selection. First, we consider selection using multiplexers; then we briefly examine selection circuits implemented by using three-state drivers or transmission gates.

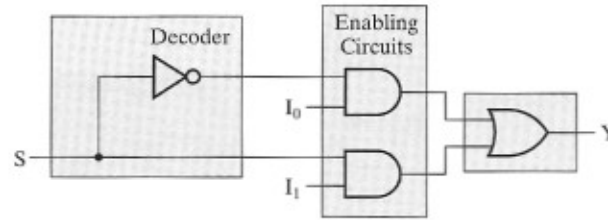
Multiplexers

A multiplexer is a combinational circuit that selects binary information from one of many input lines and directs the information to a single output line. The selection of a particular input line is controlled by a set of input variables, called *selection inputs*. Normally, there are 2^n input lines and n selection inputs whose bit combinations determine which input is selected. We begin with $n = 1$, a 2-to-1-line multiplexer. This function has two information inputs, I_0 and I_1 , and a single select input S . The truth table for the circuit is given in Table 4-6. Examining the table, if the select input $S = 0$, the output of the multiplexer takes on the values of I_0 , and, if input $S = 1$, the output of the multiplexer takes on the values of I_1 . Thus, S selects either input I_0 or input I_1 to appear at output Y . From this discussion, we can see that the equation for the 2-to-1-line multiplexer output Y is

$$Y = \bar{S} I_0 + S I_1$$

□ TABLE 4-6
Truth Table for 2-to-1-Line Multiplexer

S	I_0	I_1	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



□ **FIGURE 4-13**
A single bit 2-to-1-Line Multiplexer

This same equation can be obtained by using a 3-variable K-map. As shown in Figure 4-13, the implementation of the preceding equation can be decomposed into a 1-to-2-line decoder, two enabling circuits and a 2-input OR gate.

Suppose that we wish to design a 4-to-1-line multiplexer. In this case, the function Y depends on four inputs I_0, I_1, I_2 , and I_3 and two select inputs S_1 and S_0 . By placing the values of I_0 through I_3 in the Y column, we can form Table 4-7, a condensed truth table for this multiplexer. In this table, the information variables do not appear as input columns of the table but appear in the output column. Each row represents multiple rows of the full truth table. In Table 4-7, the row 00 I_0 represents all rows in which $(S_1, S_0) = 00$ and, for $I_0 = 1$, gives $Y = 1$ and, for $I_0 = 0$, gives $Y = 0$. Since there are six variables, and only S_1 and S_0 are fixed, this single row represents 16 rows of the corresponding full truth table. From the table, we can write the equation for Y as

$$Y = \bar{S}_1 \bar{S}_0 I_0 + \bar{S}_1 S_0 I_1 + S_1 \bar{S}_0 I_2 + S_1 S_0 I_3$$

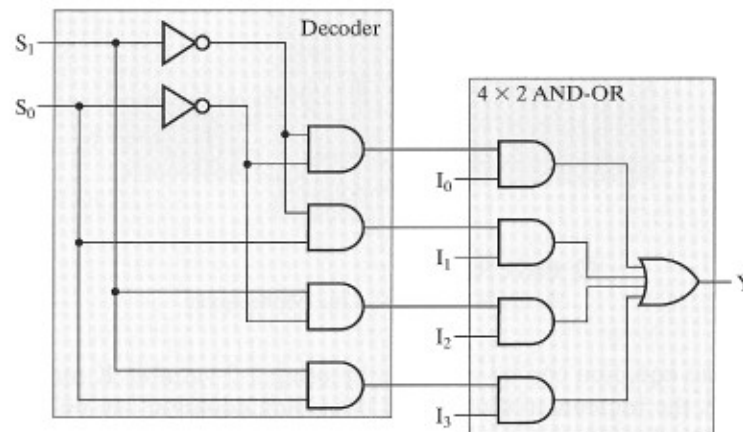
If this equation is implemented directly, two inverters, four 3-input AND gates, and a 4-input OR gate are required, giving a gate input count of 18. A different implementation can be obtained by factoring the AND terms, to give

$$Y = (\bar{S}_1 \bar{S}_0) I_0 + (\bar{S}_1 S_0) I_1 + (S_1 \bar{S}_0) I_2 + (S_1 S_0) I_3$$

This implementation can be constructed by combining a 2-to-4-line decoder, four AND gates used as enabling circuits and a 4-input OR gate as shown in Figure 4-14. We will refer to the combination of AND gates and OR gates as an

□ **TABLE 4-7**
Condensed Truth Table for 4-to-1-Line Multiplexer

S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3



□ **FIGURE 4-14**
A Single Bit 4-to-1-Line Multiplexer

$m \times 2$ AND-OR, where m is the number of AND gates and 2 is the number of inputs to the AND gates. This resulting circuit has a gate input count of 22, which is the more costly. Nevertheless, it provides a structural basis for constructing larger n -to- 2^n line multiplexers by expansion.

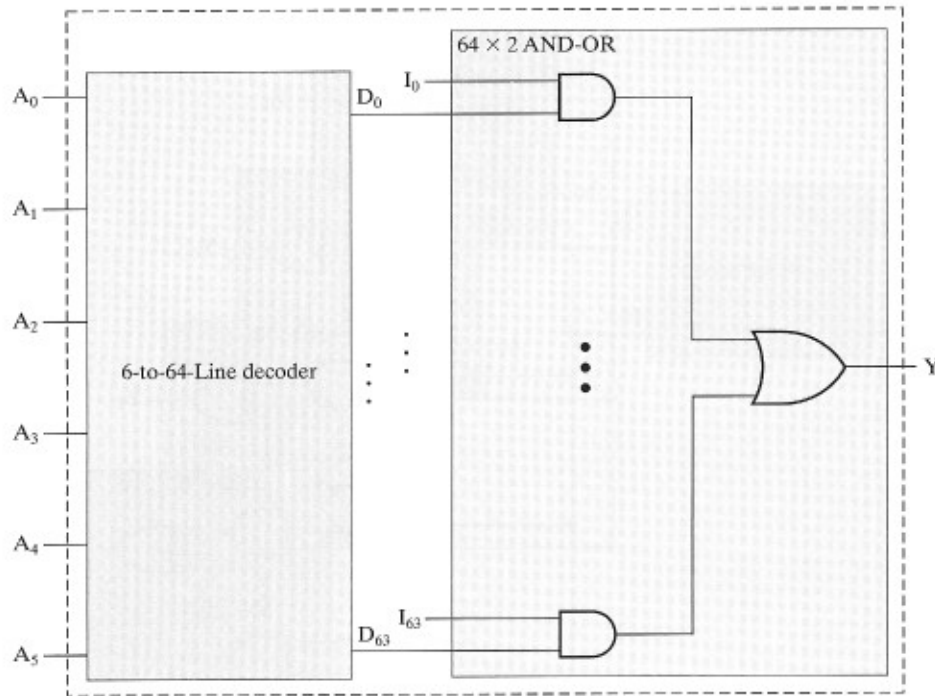
A multiplexer is also called a *data selector*, since it selects one of many information inputs and steers the binary information to the output line. The term “multiplexer” is often abbreviated as “MUX.”

Multiplexer Expansion

Multiplexers can be expanded by considering larger values of n vectors of input bits. Expansion is based upon the circuit structure given in Figure 4-14 consisting of a decoder, enabling circuits, and an OR gate. Multiplexer design is illustrated in Examples 4-4 and 4-5.

EXAMPLE 4-4 64-to-1-Line Multiplexer

A multiplexer is to be designed for $n = 6$. This will require a 6-to-64-line decoder as given in Figure 4-9, and a 64×2 AND-OR gate. The resulting structure is shown in Figure 4-15. This structure has a gate input count of $182 + 128 + 64 = 374$. In contrast, if the decoder and the enabling circuit are replaced by inverters plus 7-input AND gates, the gate input count would be $6 + 448 + 64 = 518$. For single-bit multiplexers such as this one, combining the AND gate generating D_i with the AND gate driven by D_i into a single 3-input AND gate for every $i = 0$ through 63 reduces the gate input count to 310. For multiple-bit multiplexers, this reduction to 3-input ANDs cannot be performed without replicating the output ANDs of the decoders. As a result, in almost all cases, the original structure has a



□ **FIGURE 4-15**
A 64-to-1-Line Multiplexer

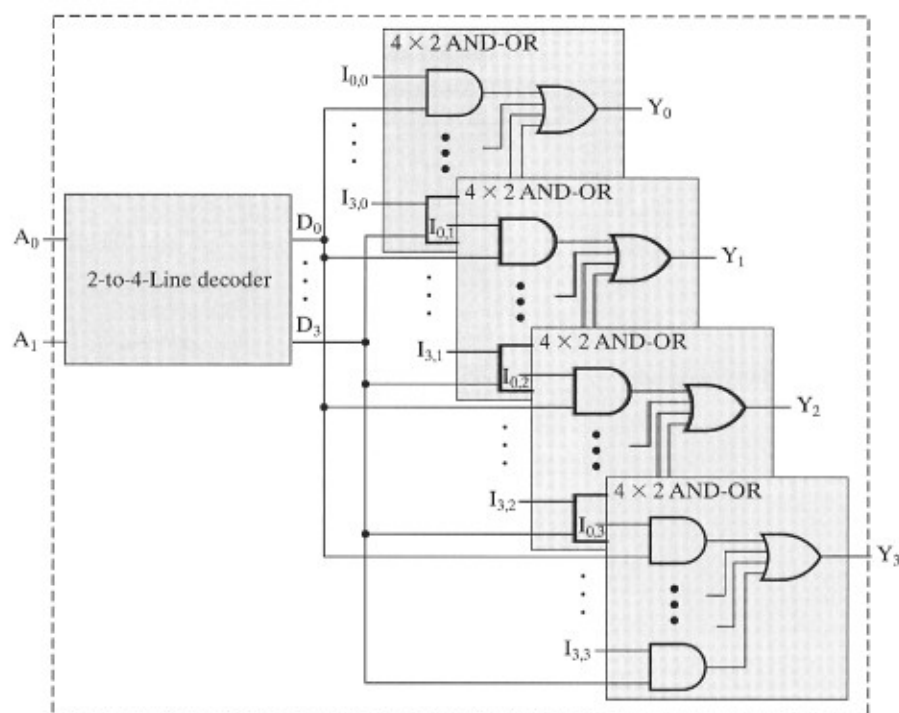
lower gate input cost. The next example illustrates the expansion to a multiple-bit multiplexer. ■

EXAMPLE 4-5 4-to-1-Line Quad Multiplexer

A quad 4-to-1-line multiplexer, which has two selection inputs and each information input replaced by a vector of four inputs, is to be designed. Since the information inputs are a vector, the output Y also becomes a four-element vector. The implementation for this multiplier requires a 2-to-4-line decoder, as given in Figure 4-7, and four 4×2 AND-OR gates. The resulting structure is shown in Figure 4-16. This structure has a gate input count of $10 + 32 + 16 = 58$. In contrast, if four 4-input multiplexers implemented with 3-input gates were placed side by side, the gate input count would be 76. So, by sharing the decoder, we reduced the gate input count. ■

Alternative Selection Implementations

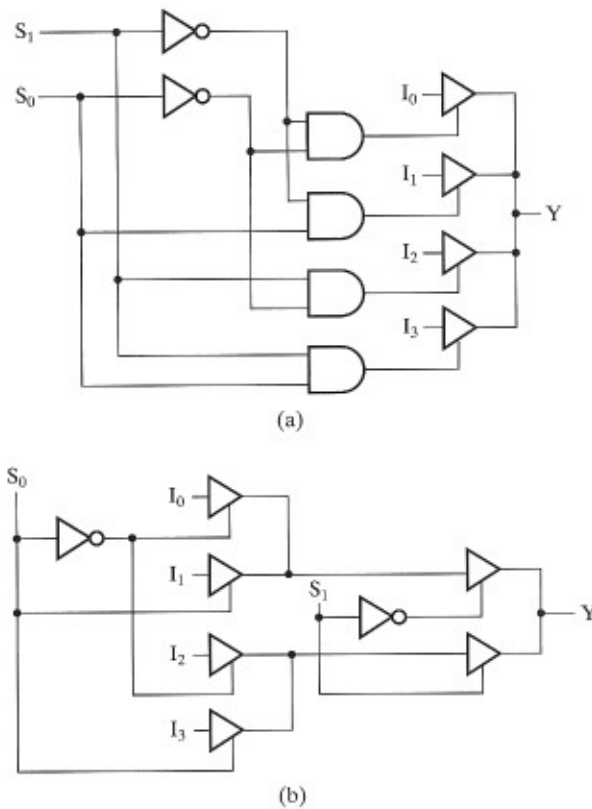
By using three-state drivers and transmission gates, it is possible to implement data selectors and multiplexers with even lower cost than is achievable with gates.



□ **FIGURE 4-16**
A Quad 4-to-1-Line Multiplexer

THREE-STATE IMPLEMENTATIONS Three-state drivers, introduced in Chapter 2, provide an alternative implementation of multiplexers. In the implementation given in Figure 4-17(a), four 3-state drivers with their outputs connected to Y replace enabling circuits and the output OR gate to give an implementation with a gate input count of 18. The logic can be reduced further by distributing the decoding across the three-state drivers as shown in Figure 4-17(b). In this case, three pairs of enabling circuits, all with two-output decoders consisting of simply a wire and an inverter, drive the enable inputs. The gate input count for this circuit is reduced to just 14.

TRANSMISSION GATE IMPLEMENTATION As a modification of the 3-state approach in Figure 4-17(b), selection circuits can be constructed with transmission gates. This implementation, shown for 4-to-1 selection in Figure 4-18, uses transmission gates as switches. The TG circuit provides a transmission path between each I input and the Y output when the two select inputs on the transmission gates on the path have the value of 1 on the terminal without a bubble and 0 on the terminal with a bubble. With the opposite value on a select input, one of the transmission gates on the

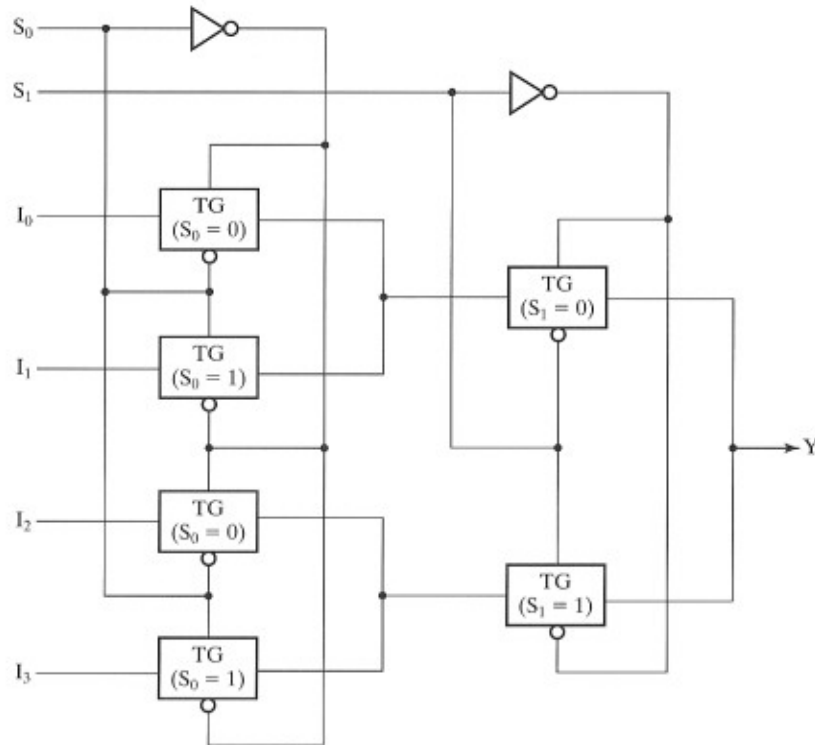


□ **FIGURE 4-17**
Selection Circuits Using Three-state Drivers

path behaves like an open switch, and the path is disconnected. The two selection inputs S_1 and S_0 control the transmission paths in the TG circuits. For example, if $S_0 = 0$ and $S_1 = 0$, there is a closed path from input I_0 to output Y , and the other three inputs are disconnected by the other TG circuits. The cost of a transmission gate is equivalent to a gate input count of one. Thus, the gate input count of this transmission gate-based multiplexer is eight.

4-6 COMBINATIONAL FUNCTION IMPLEMENTATION

Decoders and multiplexers can be used to implement Boolean functions. In addition, the programmable logic devices introduced in Chapter 3 can be viewed as containing functional blocks suitable for implementing combinational logic functions. In this section, we cover the use of decoders, multiplexers, read-only memories (ROMs), programmable logic arrays (PLAs), programmable array logic devices (PALs), and lookup tables (LUTs) for implementing combinational logic functions.



□ **FIGURE 4-18**
4-to-1-Line Multiplexer using Transmission Gates

Using Decoders

A decoder provides the 2^n minterms of n input variables. Since any Boolean function can be expressed as a sum of minterms, one can use a decoder to generate the minterms and combine them with an external OR gate to form a sum-of-minterms implementation. In this way, any combinational circuit with n inputs and m outputs can be implemented with an n -to- 2^n -line decoder and m OR gates.

The procedure for implementing a combinational circuit by means of a decoder and OR gates requires that the Boolean functions be expressed as a sum of minterms. This form can be obtained from the truth table or by plotting each function on a K-map. A decoder is chosen or designed that generates all the minterms of the input variables. The inputs to each OR gate are selected as the appropriate minterm outputs according to the list of minterms of each function. This process is shown in the next example.

EXAMPLE 4-6 Decoder and OR Gate Implementation of a Binary Adder Bit

In Chapter 1, we considered binary addition. The sum bit output S and the carry bit output C for a bit position in the addition are given in terms of the two bits being added, X and Y , and the incoming carry from the right, Z , in Table 4-8.

□ **TABLE 4-8**
Truth Table for 1-bit Binary Adder

X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

From this truth table, we obtain the functions for the combinational circuit in sum-of-minterms form:

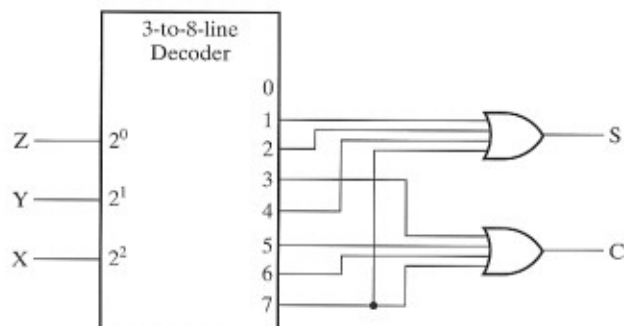
$$S(X, Y, Z) = \sum m(1, 2, 4, 7)$$

$$C(X, Y, Z) = \sum m(3, 5, 6, 7)$$

Since there are three inputs and a total of eight minterms, we need a 3-to-8-line decoder. The implementation is shown in Figure 4-19. The decoder generates all eight minterms for inputs X , Y , and Z . The OR gate for output S forms the logical sum of minterms 1, 2, 4, and 7. The OR gate for output C forms the logical sum of minterms 3, 5, 6, and 7. Minterm 0 is not used. ■

A function with a long list of minterms requires an OR gate with a large number of inputs. A function having a list of k minterms can be expressed in its complement form with $2^n - k$ minterms. If the number of minterms in a function F is greater than $2^n/2$, then the complement of F , \bar{F} , can be expressed with fewer minterms. In such a case, it is advantageous to use a NOR gate instead of an OR gate. The OR portion of the NOR gate produces the logical sum of the minterms of \bar{F} . The output bubble of the NOR gate complements this sum and generates the normal output F .

The decoder method can be used to implement any combinational circuit. However, this implementation must be compared with other possible implementations to determine the best solution. The decoder method may provide the best solution, particularly if the combinational circuit has many outputs based on the same inputs and each output function is expressed with a small number of minterms.



□ **FIGURE 4-19**
Implementing a Binary Adder Using a Decoder

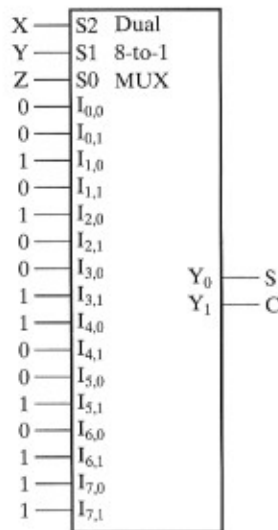
Using Multiplexers

In Section 4-5, we learned that a decoder combined with an $m \times 2$ AND-OR gate implements a multiplexer. The decoder in the multiplexer generates the minterms of the selection inputs. The AND-OR gate provides enabling circuits that determine whether the minterms are “attached” to the OR gate with the information inputs (I_i) used as the enabling signals. If I_i input is 1, then minterm m_i is attached to the OR gate, and, if the I_i input is a 0, then minterm m_i is replaced by a 0. Value-fixing applied to the I inputs provides a method for implementing a Boolean function of n variables with a multiplexer having n selection inputs and 2^n data inputs, one for each minterm. Further, an m -output function can be implemented by using value-fixing on a multiplexer with m -bit information vectors instead of the individual I bits, as illustrated by the next example.

EXAMPLE 4-7 Multiplexer Implementation of a Binary Adder Bit

The values for S and C from the 1-bit binary adder truth table given in Table 4-8 can be generated by using value-fixing on the information inputs of a multiplexer. Since there are three selection inputs and a total of eight minterms, we need a dual 8-to-1-line multiplexer for implementing the two outputs, S and C . The implementation based on the truth table is shown in Figure 4-20. Each pair of values, such as $(0, 1)$ on $(I_{1,1}, I_{1,0})$, is taken directly from the corresponding row of the last two truth table columns. ■

A more efficient method implements a Boolean function of n variables with a multiplexer that has only $n - 1$ selection inputs. The first $n - 1$ variables of the function are connected to the selection inputs of the multiplexer. The remaining variable of the function is used for the information inputs. If the final variable is Z , each data input of the multiplexer will be either Z , \bar{Z} , 1, or 0. The function can be implemented by attaching implementations of the four rudimentary functions from

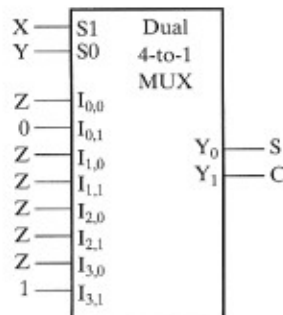


□ **FIGURE 4-20**
Implementing a 1-bit Binary Adder with a Dual 8-to-1-Line Multiplexer

Table 4-1 to the information inputs to the multiplexer. The next example demonstrates this procedure.

EXAMPLE 4-8 Alternative Multiplexer Implementation of a Binary Adder Bit

This function can be implemented with a dual 4-to-1-line multiplexer, as shown in Figure 4-21. The design procedure can be illustrated by considering the sum S . The two variables X and Y are applied to the selection lines in that order; X is connected to the S_1 input, and Y is connected to the S_0 input. The values for the data input lines are determined from the truth table of the function. When $(X, Y) = 00$, the output S is equal to Z because $S = 0$ when $Z = 0$ and $S = 1$ when $Z = 1$. This



□ **FIGURE 4-21**
Implementing a 1-bit Binary Adder with a Dual 8-to-1-Line Multiplexer

requires that the variable Z be applied to information input I_{00} . The operation of the multiplexer is such that, when $(X, Y) = 00$, information input I_{00} has a path to the output that makes S equal to Z . In a similar fashion, we can determine the required input to lines I_{10} , I_{20} , and I_{30} from the value of S when $(X, Y) = 01, 10$, and 11 , respectively. A similar approach can be used to determine the values for I_{01} , I_{11} , I_{21} , and I_{31} . ■

The general procedure for implementing any Boolean function of n variables with a multiplexer with $n - 1$ selection inputs and 2^{n-1} data inputs follows from the preceding example. The Boolean function is first listed in a truth table. The first $n - 1$ variables in the table are applied to the selection inputs of the multiplexer. For each combination of the selection variables, we evaluate the output as a function of the last variable. This function can be 0, 1, the variable, or the complement of the variable. These values are then applied to the appropriate data inputs. This process is illustrated in the next example.

EXAMPLE 4-9 Multiplexer Implementation of 4-Variable Function.

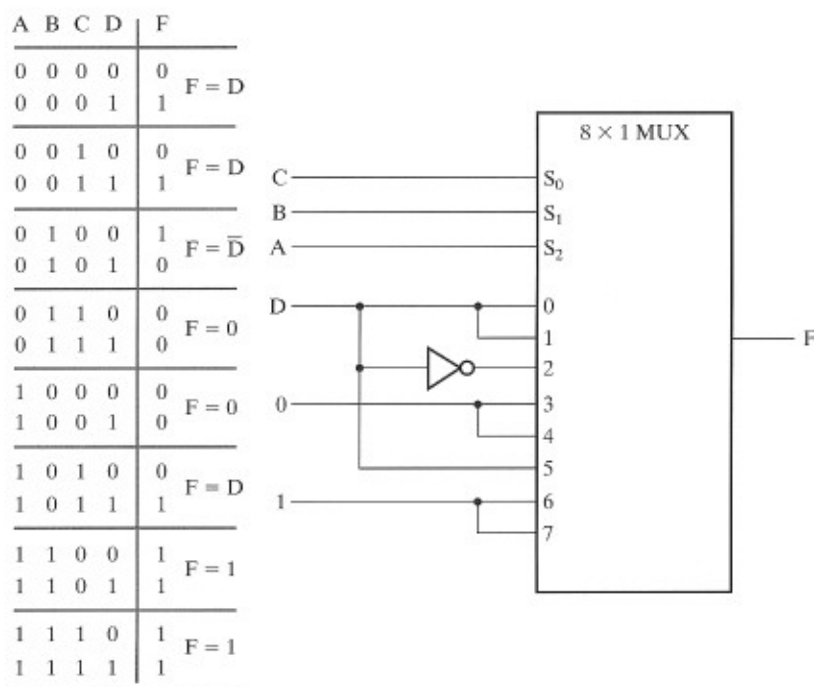
As a second example, consider the implementation of the following Boolean function:

$$F(A, B, C, D) = \sum m(1, 3, 4, 11, 12, 13, 14, 15)$$

This function is implemented with an 8×1 multiplexer as shown in Figure 4-22. To obtain a correct result, the variables in the truth table are connected to selection inputs S_2 , S_1 , and S_0 in the order in which they appear in the table (i.e., such that A is connected to S_2 , B is connected to S_1 , and C is connected to S_0 , respectively). The values for the data inputs are determined from the truth table. The information line number is determined from the binary combination of A , B , and C . For example, when $(A, B, C) = 101$, the truth table shows that $F = D$; so the input variable D is applied to information input I_5 . The binary constants 0 and 1 correspond to two fixed signal values. Recall from Section 4-2 that, in a logic schematic, these constant values are replaced by the ground and power symbols as shown in Figure 4-2. ■

Using Read-Only Memories

On the basis of the decoders and multiplexers covered thus far, there are two approaches to the implementation of read-only memories. One approach is based on a decoder and OR gates. By inserting parallel OR gates, one for each ROM output, to sum the minterms of Boolean functions, we were able to generate any desired combinational circuit. The ROM can be viewed as a device that includes both the decoder and the OR gates within a single unit. By closing connections to the inputs of an OR gate for those minterms that are included in the function, the ROM outputs can be programmed to represent the Boolean functions of the output variables in a combinational circuit. The alternative approach is based on value-fixing on a multiple-bit multiplexer. The I_i values are used as enabling



□ **FIGURE 4-22**
Implementing a Four-Input Function with a Multiplexer

signals to determine which minterms are connected to the OR gates within the multiplexer bits. This is illustrated by Example 4-8 which is equivalent to a ROM with three inputs and two outputs. The “programming” of this ROM approach is done by applying the truth table to the information inputs to the multiplexer. Because the decoder and OR gates approach is just a different model, it too can be “programmed” by using a truth table to determine the connections between the decoder and the OR gates. Thus, in practice, when a combinational circuit is designed by means of a ROM, it is not necessary to design the logic or to show the internal connections inside the unit. All that the designer has to do is specify the particular ROM by its name and provide the truth table for the ROM. The truth table gives all the information for programming the ROM. No internal logic diagram is needed to accompany the table. Example 4-10 shows this use of a ROM.

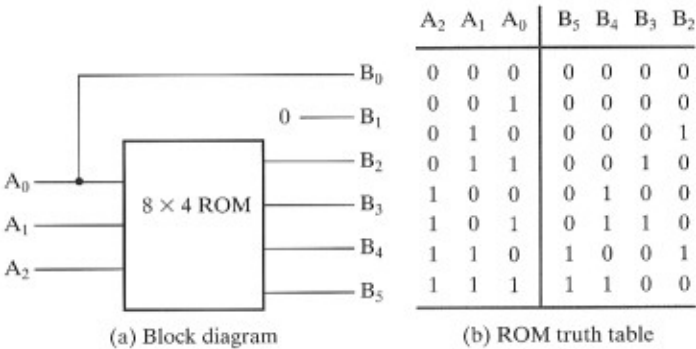
EXAMPLE 4-10 Implementing a Combinational Circuit Using a ROM

Design a combinational circuit using a ROM. The circuit accepts a 3-bit number and generates an output binary number equal to the square of the input number.

The first step in designing the circuit is to derive the truth table of the combinational circuit. In most cases, this is all that is needed. In other cases, we can use a partial truth table for the ROM by utilizing certain properties in the output variables. Table 4-9 is the truth table for the combinational circuit. Three inputs and six outputs are needed to accommodate all of the possible binary numbers. We note that output B_0 is always equal to input A_0 , so there is no need to generate B_0 with a ROM. Moreover, output B_1 is always 0, so this output is a known constant. Thus, we actually need to generate only four outputs with the ROM; the other two are readily obtained. The minimum size of ROM needed must have three inputs and four outputs. Three inputs specify eight words; so the ROM must be of size 8×4 . The implementation of the ROM is shown in Figure 4-23. The three inputs specify eight words of four bits each. The block diagram of Figure 4-23(a) shows the required connections of the combinational circuit. The truth table in Figure 4-23(b) specifies the information needed for programming the ROM. ■

□ TABLE 4-9
Truth Table for Circuit of Example 4-10

Inputs			Outputs						Decimal
A ₂	A ₁	A ₀	B ₅	B ₄	B ₃	B ₂	B ₁	B ₀	
0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	1	1
0	1	0	0	0	0	1	0	0	4
0	1	1	0	0	1	0	0	1	9
1	0	0	0	1	0	0	0	0	16
1	0	1	0	1	1	0	0	1	25
1	1	0	1	0	0	1	0	0	36
1	1	1	1	1	0	0	0	1	49



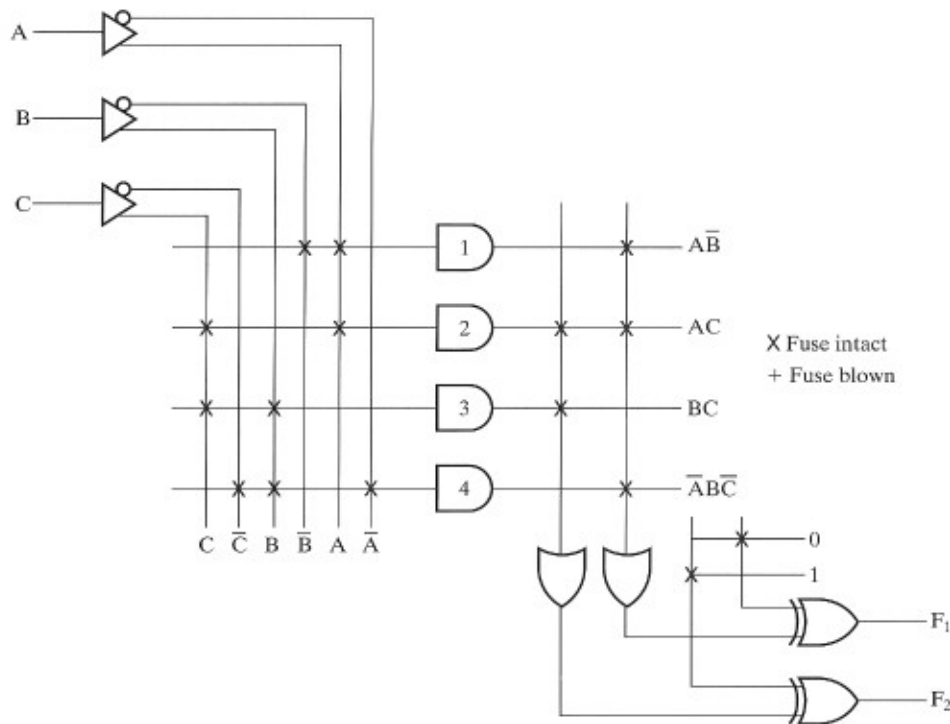
□ FIGURE 4-23
ROM Implementation of Example 4-10

ROM devices are widely used to implement complex combinational circuits directly from their truth tables. They are useful for converting from one code, such as Gray code, to another, such as BCD. They can generate complex arithmetic operations, such as multiplication or division, and in general, they are used in applications requiring a moderate number of inputs and a large number of outputs.

Using Programmable Logic Arrays

The PLA is similar in concept to the ROM, except that the PLA does not provide full decoding of the variables, so it does not generate all the minterms. The decoder is replaced by an array of AND gates, each of which can be programmed to generate any product term of the input variables. The product terms are then selectively connected to OR gates as in a ROM to provide the sum of products for the required functions.

The fuse map of a PLA can be specified in tabular form. For example, the programming table that specifies the PLA of Figure 4-24 is listed in Table 4-10. The table consists of three sections. The first section lists the product term numbers.



□ **FIGURE 4-24**
PLA with Three Inputs, Four Product Terms, and Two Outputs

□ **TABLE 4-10**
Programming Table for the PLA in Figure 4-24

		Inputs			Outputs	
		A	B	C	(T) F ₁	(C) F ₂
$\overline{A}\overline{B}$	1	1	0	—	1	—
AC	2	1	—	1	1	1
BC	3	—	1	1	—	1
$\overline{A}B\overline{C}$	4	0	1	0	1	—

The second section specifies the required paths between inputs and AND gates. The third section specifies the paths between the AND and OR gates. For each output variable, we may have a T (for true) or C (for complement) for controlling the output exclusive-OR gate. The product terms listed on the left are not part of the table; they are included for reference only. For each product term, the inputs are marked with 1, 0, or — (dash). If a variable in the product term appears in its true form, the corresponding input variable is marked with a 1. If the variable in the product term appears complemented, the corresponding input variable is marked with a 0. If the variable is absent in the product term, it is marked with a dash.

The paths between the inputs and the AND gates are specified under the column heading “Inputs” in the table. A 1 in the input column specifies a CLOSED circuit from the input variable to the AND gate. A 0 in the input column specifies a CLOSED circuit from the complement of the variable to the input of the AND gate. A dash specifies OPEN circuits for both the input variable and its complement. It is assumed that an OPEN terminal on the input of an AND gate behaves like a 1.

The paths between the AND and OR gates are specified under the column heading “Outputs.” The output variables are marked with 1’s for those product terms that are included in the function. Each product term that has a 1 in the output column requires a CLOSED path from the output of the AND gate to the input of the OR gate. Those product terms marked with a dash specify an OPEN circuit. It is assumed that an open terminal on the input of an OR gate behaves like a 0. Finally, a T (true) output dictates that the other input of the corresponding XOR gate be connected to 0, and a C (complement) specifies a connection to 1.

The size of a PLA is specified by the number of inputs, the number of product terms, and the number of outputs. A typical PLA has 16 inputs, 48 product terms, and eight outputs. For n inputs, k product terms, and m outputs, the internal logic of the PLA consists of n buffer-inverter gates, k AND gates, m OR gates, and m XOR gates. There are $2n \times k$ programmable connections between the inputs and the AND array, $k \times m$ programmable connections between the AND and OR arrays, and m programmable connections associated with the XOR gates.

In designing a digital system with a PLA, there is no need to show the internal connections of the unit, as was done in Figure 4-24. All that is needed is a PLA programming table from which the PLA can be programmed to supply the required logic. As with a ROM, the PLA may be mask programmable or field programmable.

In implementing a combinational circuit with a PLA, a careful investigation must be undertaken in order to reduce the number of distinct product terms, so that the complexity of the circuit may be reduced. Fewer product terms can be achieved by simplifying the Boolean function to a minimum number of terms. The number of literals in a term is less important, since all the input variables are available anyway. It is wise, however, to avoid extra literals, as these may cause problems in testing the circuit and may reduce the speed of the circuit. Both the true and complement forms of each function should be simplified to see which one can be expressed with fewer product terms and which one provides product terms that are common to other functions. This process is shown in Example 4-11.

EXAMPLE 4-11 Implementing a Combinational Circuit Using a PLA

Implement the following two Boolean functions with a PLA:

$$F_1(A, B, C) = \Sigma m(0, 1, 2, 4)$$

$$F_2(A, B, C) = \Sigma m(0, 5, 6, 7)$$

The two functions are simplified in the maps of Figure 4-25. Both the true and complement outputs of the functions are simplified in sum-of-products form. The combination that gives a minimum number of product terms is

$$F_1 = \overline{AB} + AC + BC$$

$$F_2 = AB + AC + \overline{A}\overline{B}\overline{C}$$

The simplification gives four distinct product terms: $\overline{A}\overline{B}$, AC , BC , and $\overline{A}\overline{B}\overline{C}$. The PLA programming table for this combination is shown in the figure. Note that output F_1 is the true output, even though a \overline{C} is marked over it in the table. This is because \overline{F}_1 is generated with an AND-OR circuit and is available at the output of the OR gate. The XOR gate complements the function \overline{F}_1 to produce the true F_1 output. ■

Using Programmable Array Logic Devices

In designing with a PAL device, the Boolean functions must be simplified to fit into each section as illustrated by the example PAL device in Figure 4-26. Unlike the arrangement in the PLA, a product term cannot be shared among two or more OR gates. Therefore, each function can be simplified by itself, without regard to

		B					
		BC		00	01	11	10
A	0	1	1	0	1		
	1	1	0	0	0		
		C					
		$F_1 = \overline{A}\overline{B} + \overline{A}\overline{C} + \overline{B}\overline{C}$					
		$\overline{F}_1 = AB + AC + BC$					

		B					
		BC		00	01	11	10
A	0	1	0	0	0		
	1	0	1	1	1		
		C					
		$F_2 = AB + AC + \overline{A}\overline{B}\overline{C}$					
		$\overline{F}_2 = \overline{A}\overline{C} + \overline{A}\overline{B} + \overline{A}\overline{B}\overline{C}$					

PLA programming table

	Product term	Outputs					
		Inputs (C)			(T)		
		A	B	C	F_1	F_2	
AB	1	1	1	-	1	1	
AC	2	1	-	1	1	1	
BC	3	-	1	1	1	-	
$\overline{A}\overline{B}\overline{C}$	4	0	0	0	-	1	

□ FIGURE 4-25

Solution to Example 4-11

common product terms. The number of product terms in each section is fixed, and if the number of terms in the function is too large, it may be necessary to use two or more sections to implement one Boolean function. In such a case, common terms may be useful. This process is illustrated in Example 4-12.

EXAMPLE 4-12 Implementing a Combinational Circuit Using a PAL

As an example of a PAL device incorporated into the design of a combinational circuit, consider the following Boolean functions, given in sum-of-minterms form:

$$W(A, B, C, D) = \Sigma m(2, 12, 13)$$

$$X(A, B, C, D) = \Sigma m(7, 8, 9, 10, 11, 12, 13, 14, 15)$$

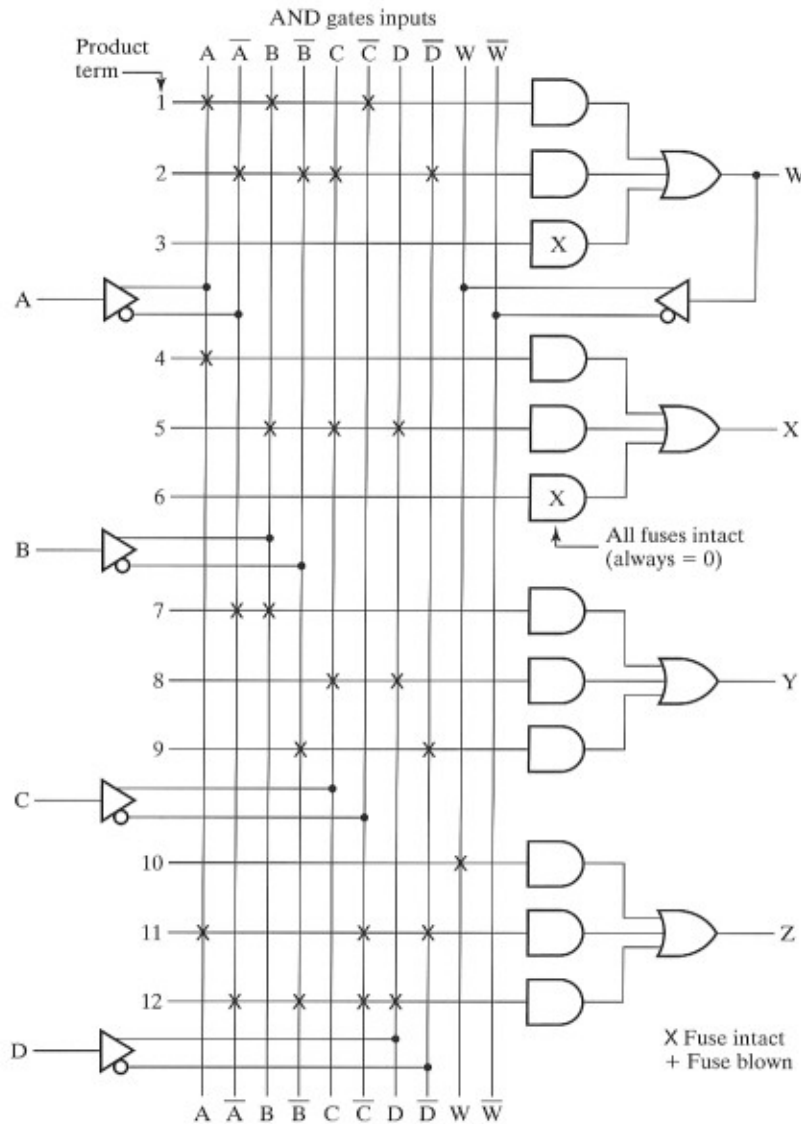
$$Y(A, B, C, D) = \Sigma m(0, 2, 3, 4, 5, 6, 7, 8, 10, 11, 15)$$

$$Z(A, B, C, D) = \Sigma m(1, 2, 8, 12, 13)$$

Simplifying the four functions to a minimum number of terms results in the following Boolean functions:

$$W = AB\overline{C} + \overline{A}\overline{B}C\overline{D}$$

$$X = A + BCD$$



□ **FIGURE 4-26**
Connection Map for PAL[®] Device for Example 4-12

$$Y = \overline{A}B + CD + \overline{B}D$$

$$Z = ABC\overline{C} + \overline{A}\overline{B}C\overline{D} + A\overline{C}\overline{D} + \overline{A}\overline{B}\overline{C}D$$

$$= W + A\overline{C}\overline{D} + \overline{A}\overline{B}\overline{C}D$$

Note that the function for Z has four product terms. The logical sum of two of these terms is equal to W . Thus, by using W , it is possible to reduce the number of terms for Z from four to three, so that the functions can fit into the PAL device in Figure 4-26.

The PAL programming table is similar to the table used for the PLA, except that only the inputs of the AND gates need to be programmed. Table 4-11 lists the PAL programming table for the preceding four Boolean functions. The table is divided into four sections with three product terms in each, to conform with the PAL device of Figure 4-26. The first two sections need only two product terms to implement the Boolean function. By placing W in the first section of the device, the feedback connection from $F1$ into the array is available to reduce the function Z to three terms.

The connection map for the PAL device, as specified in the programming table, is shown in Figure 4-26. For each 1 or 0 in the table, we mark the corresponding intersection in the diagram with the symbol for a CLOSED circuit connection. For each dash, we mark the diagram with OPEN circuit connections for both the true and complement inputs. If the AND gate is not used, we leave all of its inputs as CLOSED circuits. Since the corresponding input receives both the true and the complement of each input variable, we have $A \cdot \overline{A} = 0$, and the output of the AND gate is always 0. ■

□ TABLE 4-11
PAL® Programming Table for Example 4-12

Product term	AND Inputs					Outputs
	A	B	C	D	W	
1	1	1	0	—	—	$W = \overline{A}B\overline{C}$ $+ \overline{A}\overline{B}C\overline{D}$
2	0	0	1	0	—	
3	—	—	—	—	—	
4	1	—	—	—	—	$X = A$ $+ BCD$
5	—	1	1	1	—	
6	—	—	—	—	—	
7	0	1	—	—	—	$Y = \overline{A}B$ $+ CD$ $+ \overline{B}D$
8	—	—	1	1	—	
9	—	0	—	0	—	
10	—	—	—	—	1	$Z = W$ $+ A\overline{C}\overline{D}$ $+ \overline{A}\overline{B}\overline{C}D$
11	1	—	0	0	—	
12	0	0	0	1	—	

Using Lookup Tables

Field-programmable gate arrays (FPGAs) and complex programmable logic devices (CPLDs), often use lookup tables (LUTs) for implementing logic. Programming a single function with m inputs is the same as programming a single output ROM (i.e., the lookup table simply stores the truth table of the function). An m -input lookup table can implement any function of m or fewer variables. Typically, $m = 4$. The key problem in programming lookup tables is dealing with functions with $> m$ input variables. Also, sharing of lookup tables among multiple functions is important. These problems can be dealt with using multiple-level logic transformations, primarily decomposition and extraction. The optimization goal is to implement the function or functions by using a minimum number of LUTs with the constraint that a single LUT can implement a subfunction of at most m variables. This can be accomplished by finding a minimum number of equations, each with at most m variables, that implement the desired function or functions. This process is illustrated for single-output and multiple-output functions with $m = 4$ in the following examples.

EXAMPLE 4-13 Implementing a Single-Output Function with Lookup Tables

Implement the following Boolean function with lookup tables:

$$F_1(A, B, C, D, E, F, G, H, I) = ABCDE + \overline{F}GHI\overline{D}\overline{E}$$

The number of input variables for a function is called the *support* s of the function. The support for F_1 is $s = 9$. It is apparent that the minimum number of lookup tables k needed is at least $9/4$ (i.e., $k = 3$). Further, for an m -output function, the minimum number of lookup tables for a single-output function must obey an even more stringent relationship, $mk \geq s + k - 1$, so k must satisfy $4k \geq 9 + k - 1$. Solving, $k = 3$, so we will look for a decomposition of F_1 into three equations, each with at most $s = 4$. Factoring F_1 , we obtain

$$F_1 = (ABC)DE + (\overline{F}GHI)\overline{D}\overline{E}$$

Based on this equation, F_1 can be decomposed into three equations with $s \leq 4$:

$$F_1(D, E, X_1, X_2) = X_1DE + X_2\overline{D}\overline{E}$$

$$X_1(A, B, C) = ABC$$

$$X_2(F, G, H, I) = \overline{F}GHI$$

Each of these three equations can be implemented by one LUT, giving an optimum implementation for F_1 . ■

EXAMPLE 4-14 Implementing a Multiple-Output Function with Lookup Tables

Implement the following pair of Boolean function with lookup tables:

$$F_1(A, B, C, D, E, F, G, H, I) = ABCDE + \overline{F}GHI\overline{D}\overline{E}$$

$$F_2(A, B, C, D, E, F, G, H, I) = ABCEF + \overline{F}GHI$$

Each of these functions requires a support $s = 9$. Thus, at least three LUTs are required for each function. But two of the LUTs can be shared, so the minimum number of LUTs needed is $k = 6 - 2 = 4$. Factoring F_2 with sharing of equations with the decomposition of F_1 from the previous example gives

$$F_2 = (ABC)EF + (\overline{F}GHI)$$

This produces an extraction for F_1 and F_2 :

$$F_1(D, E, X_1, X_2) = X_1DE + X_2\overline{D}\overline{E}$$

$$F_2(E, F, X_1, X_2) = X_1EF + X_2$$

$$X_1(A, B, C) = ABC$$

$$X_2(F, G, H, I) = \overline{F}GHI$$

In this case, the extraction requires four LUTs, the minimum number. Note that in general, there is no guarantee that a decomposition or extraction can be found that requires the minimum number of LUTs calculated. ■

4-7 HDL REPRESENTATION FOR COMBINATIONAL CIRCUITS—VHDL

Since an HDL is used for describing and designing hardware, it is very important to keep the underlying hardware in mind as you write in the language. This is particularly critical if your language description is to be synthesized. For example, if you ignore the hardware that will be generated, it is very easy to specify a large complex gate structure by using \times (multiply) when a much simpler structure using only a few gates is all that is needed. For this reason, we initially emphasize description of detailed hardware with VHDL, and proceed to more abstract, higher-level descriptions later.

Selected examples in this chapter are useful for introducing VHDL as an alternative means for representing detailed digital circuits. Initially, we show structural VHDL descriptions that replace the schematic for the 2-to-4-line decoder with enable given in Figure 4-10 on page 152. This example and one using the 4-to-1-line multiplexer in Figure 4-14 on page 158 illustrate many of the

fundamental concepts of VHDL. We then present higher level functional and behavioral VHDL descriptions for these circuits that further illustrate fundamental VHDL concepts.

EXAMPLE 4-15 Structural VHDL for a 2-to-4-Line Decoder

Figure 4-27 shows a VHDL description for the 2-to-4-line decoder circuit from Figure 4-10 on page 152. This example will be used to demonstrate a number of general VHDL features as well as structural description of circuits.

The text between two dashes -- and the end of the line is interpreted as a *comment*. So the description in Figure 4-27 begins with a two-line comment identifying the description and its relationship to Figure 4-10. To assist in discussion of

```
-- 2-to-4 Line Decoder with Enable: Structural VHDL Description      -- 1
-- (See Figure 4-10 for logic diagram)                             -- 2
library ieee, lcdf_vhdl;                                           -- 3
use ieee.std_logic_1164.all, lcdf_vhdl.func_prims.all;            -- 4
entity decoder_2_to_4_w_enable is                                   -- 5
    port (EN, A0, A1: in std_logic;                                -- 6
          D0, D1, D2, D3: out std_logic);                          -- 7
end decoder_2_to_4_w_enable;                                       -- 8
                                                                    -- 9

architecture structural_1 of decoder_2_to_4_w_enable is            -- 10
    component NOT1                                                -- 11
        port (in1: in std_logic;                                  -- 12
              out1: out std_logic);                               -- 13
    end component;                                                -- 14
    component AND2                                                 -- 15
        port (in1, in2: in std_logic;                             -- 16
              out1: out std_logic);                               -- 17
    end component;                                                -- 18
    signal A0_n, A1_n, N0, N1, N2, N3: std_logic;                 -- 19
begin                                                            -- 20
    g0: NOT1 port map (in1 => A0, out1 => A0_n);                  -- 21
    g1: NOT1 port map (in1 => A1, out1 => A1_n);                  -- 22
    g2: AND2 port map (in1 => A0_n, in2 => A1_n, out1 => N0);      -- 23
    g3: AND2 port map (in1 => A0, in2 => A1_n, out1 => N1);       -- 24
    g4: AND2 port map (in1 => A0_n, in2 => A1, out1 => N2);       -- 25
    g5: AND2 port map (in1 => A0, in2 => A1, out1 => N3);         -- 26
    g6: AND2 port map (in1 => EN, in2 => N0, out1 => D0);         -- 27
    g7: AND2 port map (in1 => EN, in2 => N1, out1 => D1);         -- 28
    g8: AND2 port map (in1 => EN, in2 => N2, out1 => D2);         -- 29
    g9: AND2 port map (in1 => EN, in2 => N3, out1 => D3);         -- 30
end structural_1;                                                 -- 31
```

□ FIGURE 4-27
Structural VHDL Description of 2-to-4-Line Decoder

this description, comments providing line numbers have been added on the right. As a language, VHDL has a syntax that describes precisely the valid constructs that can be used in the language. This example will illustrate many aspects of the syntax. In particular, note the use of semicolons, commas and colons in the description.

Initially, we skip lines 3 and 4 of the description to focus on the overall structure. Line 5 begins the declaration of an *entity*, which is the fundamental unit of a VHDL design. In VHDL, just as for a symbol in a schematic, we need to give the design a name and to define its inputs and outputs. This is the function of the *entity declaration*. **Entity** and **is** are keywords in VHDL. Keywords, which we show in bold type, have a special meaning and cannot be used to name objects such as entities, inputs, outputs or signals. Statement **entity** decoder_2_to_4_w_enable **is** declares that a design exists with the name decoder_2_to_4_w_enable. VHDL is case insensitive (i.e., names and keywords are not distinguished by the use of uppercase or lowercase letters). DECODER_2_4_W_ENABLE is the same as Decoder_2_4_w_Enable and decoder_2_4_w_enable.

Next, a *port declaration* in lines 6 and 7 is used to define the inputs and outputs just as we would do for a symbol in a schematic. For the example design, there are three input signals: EN, A0, and A1. The fact that these are inputs is denoted by the mode **in**. Likewise, D0, D1, D2 and D3 are denoted as outputs by the mode **out**. VHDL is a strongly-typed language, so the type of the inputs and output must be declared. In this case, the type is *std_logic*, which represents *standard logic*. This type declaration specifies the values that may appear on the inputs and the outputs, as well as the operations that may be applied to the signals. Standard logic, among its nine values, includes the usual binary values 0 and 1 and two additional values X and U. X represents an unknown value, U an uninitialized value. We have chosen to use standard logic, which includes these values, since these values are used by typical simulation tools.

In order to use the type *std_logic*, it is necessary to define the values and the operations. For convenience, a *package* consisting of precompiled VHDL code is employed. Packages are usually stored in a directory referred to as a *library*, which is shared by some or all of the tool users. For *std_logic*, the basic package is *ieee.std_logic_1164*. This package defines the values and basic logic operators for types *std_ulogic* and *std_logic*. In order to use *std_logic*, we include line 3 to call up the **library** of packages called *ieee* and include line 4 containing *ieee.std_logic_1164.all* to indicate we want to use **all** of the package *std_logic_1164* from the *ieee* library. An additional library, *lcdf_vhdl*, contains a package called *func_prims* made up of basic logic gates, latches and flip-flops described using VHDL, of which we use **all**. Library *lcdf_vhdl* is available in ASCII for copying from the Prentice Hall Companion Website for the text. Note that the statements in lines 3 and 4 are tied to the entity that follows. If another entity is included that uses type *std_logic* and the elements from *func_prims*, these statements must be repeated prior to that entity declaration.

The entity declaration ends with keyword **end** followed by the entity name. Thus far, we have discussed the equivalent of a schematic symbol in VHDL for the circuit.



STRUCTURAL DESCRIPTION Next, we want to specify the function of the circuit. A particular representation of the function of an entity is called the *architecture* of the entity. Thus, the contents of line 10 declare a VHDL architecture named `structural_1` for the entity `decoder_2_to_4_w_enable` to exist. The details of the architecture follow. In this case, we use a *structural description* that is equivalent to the schematic for the circuit given in Figure 4-10.

First, we declare the gate types we are going to use as components of our description in lines 11 through 18. Since we are building this architecture from gates, we declare an inverter called `NOT1` and a 2-input AND gate called `AND2` as *components*. These gate types are VHDL descriptions in package `func_prims` that contain the entity and architecture for each of the gates. The name and the port declaration for a component must be identical to those for the underlying entity. For `NOT1`, `port` gives the input name `in1` and the output name `out1`. The component declaration for `AND2` gives input names `in1` and `in2`, and output name `out1`.

Next, before specifying the interconnection of the gates, which is equivalent to a circuit netlist, we need to name all of the nets in the circuit. The inputs and outputs already have names. The internal nets are the outputs of the two inverters and of the leftmost four AND gates in Figure 4-10. These output nets are declared as *signals* of type `std_logic`. `A0_n` and `A1_n` are the signals for the two inverter outputs and `N0`, `N1`, `N2`, and `N3` are the signals for the four AND gate outputs. Likewise, all of the inputs and outputs declared as ports are signals. In VHDL, there are both signals and variables. Variables are evaluated instantaneously. In contrast, signals are evaluated at some future point in time. This time may be physical time, such as 2 ns from the current time, or may be what is called *delta time*, in which a signal is evaluated one delta time from the current time. Delta time is viewed as an infinitesimal amount of time. Some time delay in evaluation of signals is essential to the internal operation of the typical digital simulator and, of course, based on the delay of gates, is realistic in performing simulations of circuits. For simplicity, we will typically be simulating circuits for correct function, not for performance or delay problems. For such functional simulation, it is easiest to let the delays default to delta times. Thus, no delay will be explicit in our VHDL descriptions of circuits, although delays may appear in test benches.

Following the declaration of the internal signals, the main body of the architecture starts with the keyword **begin**. The circuit described consists of two inverters and eight 2-input AND gates. Line 21 gives the label `g0` to the first inverter and indicates that the inverter is component `NOT1`. Next is a **port map**, which maps the input and output of the inverter to the signals to which they are connected. This particular form of port map uses `=>` with the port of the gate on the left and the signal to which it is connected on the right. For example, the input of inverter `g0` is `A0` and the output is `A0_n`. Lines 22 through 30 give the remaining nine gates and the signals connected to their inputs and outputs. For example, in line 24, `A0` and `A1_n` are inputs and `N1` is the output. The architecture is completed with the keyword **end** followed by its name `structural_1`. ■

EXAMPLE 4-16 Structural VHDL for a 4-to-1 Multiplexer

In Figure 4-28, the structural description of the 4-to-1-line multiplexer from Figure 4-14 on page 158 illustrates two additional VHDL concepts: *std_logic_vector* and an alternative approach to mapping ports

In lines 6 and 7, instead of specifying *S* and *I* as individual *std_logic* inputs, they are specified as *std_logic_vectors*. In specifying vectors, we use an index. Since

```

-- 4-to-1 Line Multiplexer: Structural VHDL Description           -- 1
-- (See Figure 4-14 for logic diagram)                           -- 2
library ieee, lcdf_vhdl;                                         -- 3
use ieee.std_logic_1164.all, lcdf_vhdl.func_prims.all;          -- 4
entity multiplexer_4_to_1_st is                                   -- 5
    port (S: in std_logic_vector(0 to 1);                        -- 6
          I: in std_logic_vector(0 to 3);                        -- 7
          Y: out std_logic);                                     -- 8
end multiplexer_4_to_1_st;                                       -- 9
                                                                --10
architecture structural_2 of multiplexer_4_to_1_st is           --11
    component NOT1                                              --12
        port(in1: in std_logic;                                  --13
              out1: out std_logic);                              --14
    end component;                                              --15
    component AND2                                              --16
        port(in1, in2: in std_logic;                             --17
              out1: out std_logic);                              --18
    end component;                                              --19
    component OR4                                               --20
        port(in1, in2, in3, in4: in std_logic;                   --21
              out1: out std_logic);                              --22
    end component;                                              --23
    signal S_n: std_logic_vector(0 to 1);                       --24
    signal D, N: std_logic_vector(0 to 3);                       --25
begin                                                           --26
    g0: NOT1 port map (S(0), S_n(0));                           --27
    g1: NOT1 port map (S(1), S_n(1));                           --28
    g2: AND2 port map (S_n(1), S_n(0), D(0));                   --29
    g3: AND2 port map (S_n(1), S(0), D(1));                     --30
    g4: AND2 port map (S(1), S_n(0), D(2));                     --31
    g5: AND2 port map (S(1), S(0), D(3));                       --32
    g6: AND2 port map (D(0), I(0), N(0));                       --33
    g7: AND2 port map (D(1), I(1), N(1));                       --34
    g8: AND2 port map (D(2), I(2), N(2));                       --35
    g9: AND2 port map (D(3), I(3), N(3));                       --36
    g10: OR4 port map (N(0), N(1), N(2), N(3), Y);              --37
end structural_2;                                              --38

```

□ **FIGURE 4-28**

Structural VHDL Description of 4-to-1-Line Multiplexer

S consists of two input signals numbered 0 and 1, the index for *S* is 0 to 1. The components of this vector are *S*(0) and *S*(1). *I* consists of four input signals numbered 0 through 3, so the index for *I* is 0 to 3. Likewise, in lines 24 and 25, we specify signals *S_n*, *D*, and *N* as *std_logic_vectors*. *D* represents the decode outputs, and *N* represents the four internal signals between the AND gates and the OR gate.

Beginning at line 27, note how the signals within *std_logic_vectors* are referred to by giving the signal name and the index in parentheses. It is also possible to refer to subvectors (e.g., *N*(1 to 2)), which refers to *N*(1) and *N*(2), the center two signals in *N*). Also, if one wishes to have the larger index for a vector appear first, VHDL uses a somewhat different notational approach. For example, signal *N*: *std_logic_vector*(3 downto 0) defines the first bit in signal *N* as *N*(3) and the last signal in *N* as *N*(0).

In lines 27 through 37, an alternative method is used to specify the port maps for the logic gates. Instead of explicitly giving the component input and output names, we assume that these names are in the port map in the same order as given for the component. We can then implicitly specify the signals attached to these names by listing the signals in same order as the names. For example, in line 29, *S_n*(1) appears first, and so is connected to *in1*. *S_n*(0) appears second, and so is connected to *in2*. Finally, *D*(0) is connected to *out1*.

Otherwise, this VHDL description is similar in structure to that for the 2-to-4-line decoder, except that the schematic represented is that in Figure 4-14 on page 158.

DATAFLOW DESCRIPTION A dataflow description describes a circuit in terms of function rather than structure and is made up of concurrent assignment statements or their equivalent. Concurrent assignment statements are executed concurrently (i.e., in parallel) whenever one of the values on the right-hand side of the statement changes. For example, whenever a change occurs in a value on the right-hand side of a Boolean equation, the left-hand side is evaluated. The use of dataflow descriptions made up of Boolean equations is illustrated in Example 4-17.

EXAMPLE 4-17 Dataflow VHDL for a 2-to-4-Line Decoder

Figure 4-29 shows a VHDL description for the 2-to-4 line decoder circuit from Figure 4-10 on page 152. This example will be used to demonstrate a dataflow description made up of Boolean equations. The library, use, and entity statements are identical to those in Figure 4-27, so they are not repeated here. The dataflow description begins in line 9. The signals *A0_n* and *A1_n* are defined by signal assignments that apply the **not** operation to the input signal *A0* and *A1*, respectively. In line 11, *A0_n*, *A1_n* and *EN* are combined with an **and** operator to form *D0*. *D1*, *D2*, and *D3* are similarly defined in lines 12 through 14. Note that this dataflow description is much simpler than the structural description in Figure 4-27.

```

-- 2-to-4 Line Decoder: Dataflow VHDL Description          -- 1
-- (See Figure 4-10 for logic diagram)                    -- 2
Use library, use, and entity entries from 2_to_4_decoder_st; -- 3
-- 4
architecture dataflow_1 of decoder_2_to_4_w_enable is    -- 5
-- 6
signal A0_n, A1_n: std_logic;                            -- 7
begin                                                    -- 8
    A0_n <= not A0;                                       -- 9
    A1_n <= not A1;                                       -- 10
    D0 <= A0_n and A1_n and EN;                          -- 11
    D1 <= A0 and A1_n and EN;                          -- 12
    D2 <= A0_n and A1 and EN;                          -- 13
    D3 <= A0 and A1 and EN;                            -- 14
end dataflow_1;                                         -- 15

```

□ **FIGURE 4-29**

Dataflow VHDL Description of 2-to-4-Line Decoder

In the next two examples, we describe the 4-to-1-line multiplexer to illustrate two alternative forms of data flow description: when-else and with-select.

EXAMPLE 4-18 VHDL for a 4-to-1-Line Multiplexer Using When-Else

In Figure 4-30, instead of using Boolean equation-like statements in the architecture to describe the multiplexer, we use a *when-else* statement. This statement is a

```

-- 4-to-1 Line Mux: Conditional Dataflow VHDL Description -- 1
-- Using When-Else (See Table 4-7 for function table)    -- 2
library ieee;                                           -- 3
use ieee.std_logic_1164.all;                            -- 4
entity multiplexer_4_to_1_we is                        -- 5
    port (S : in std_logic_vector(1 downto 0);          -- 6
          I : in std_logic_vector(3 downto 0);          -- 7
          Y : out std_logic);                            -- 8
end multiplexer_4_to_1_we;                              -- 9
-- 10
architecture function_table of multiplexer_4_to_1_we is -- 11
begin                                                    -- 12
    Y <= I(0) when S = "00" else                        -- 13
        I(1) when S = "01" else                        -- 14
        I(2) when S = "10" else                        -- 15
        I(3) when S = "11" else                        -- 16
        'X';                                              -- 17
end function_table;                                     -- 18

```

□ **FIGURE 4-30**

Conditional Dataflow VHDL Description of 4-to-1-Line Multiplexer Using When-Else

representation of the function table given as Table 4-7 on page 157. When S takes on a particular binary value, then a particular input $I(i)$ is assigned to output Y . When the value on S is 00, then Y is assigned $I(0)$. Otherwise, the **else** is invoked so that when the value on S is 01, then Y is assigned $I(1)$, and so on. In standard logic, each of the bits can take on 9 different values. So the pair of bits for S can take on 81 possible values, only 4 of which have been specified so far. In order to define Y for the remaining 77 values, the final **else** followed by X (unknown) is given. This assigns the value X to Y if any of these 77 values occurs on S . This output value occurs only in simulation, however, since Y will always take on a 0 or 1 value in an actual circuit. ■

EXAMPLE 4-19 VHDL for a 4-to-1-Line Multiplexer Using With-Select

With-select is a variation on when-else as illustrated for the 4-to-1-line multiplexer in Figure 4-31. The expression, the value of which is to be used for the decision, follows **with** and precedes **select**. The values for the expression that causes the alternative assignments then follow **when** with each of the assignment-value pairs separated by commas. In the example, S is the signal, the value of which determines the value selected for Y . When $S = "00"$, $I(0)$ is assigned to Y . When $S = "01"$, $I(1)$ is assigned to Y and so on. ' X ' is assigned to Y **when others**, where **others** represents the 77 standard logic combinations not already specified. ■

```
--4-to-1 Line Mux: Conditional Dataflow VHDL Description      -- 1
Using with Select (See Table 4-7 for function table)          -- 2
library ieee;                                                 -- 3
use ieee.std_logic_1164.all;                                  -- 4
entity multiplexer_4_to_1_ws is                                -- 5
    port (S : in std_logic_vector(1 downto 0);                -- 6
          I : in std_logic_vector(3 downto 0);                -- 7
          Y : out std_logic);                                  -- 8
end multiplexer_4_to_1_ws;                                     -- 9
                                                                -- 10
architecture function_table_ws of multiplexer_4_to_1_ws is    -- 11
begin                                                         -- 12
    with S select                                              -- 13
        Y <= I(0) when "00",                                   -- 14
            I(1) when "01",                                   -- 15
            I(2) when "10",                                   -- 16
            I(3) when "11",                                   -- 17
            'X' when others;                                   -- 18
end function_table_ws;                                         -- 19
```

□ FIGURE 4-31

Conditional Dataflow VHDL Description of 4-to-1-Line Multiplexer Using With-Select

Note that when-else permits decisions on multiple distinct signals. For example, for the demultiplexer in Figure 4-10, the first **when** can be conditioned on input **EN** with the subsequent **when**'s conditioned on input **S**. In contrast, the with-select can depend on only a single Boolean condition (e.g., either **EN** or **S**, but not both). Also, for typical synthesis tools, when-else typically results in a more complex logical structure, since each of the decisions depends not only on the condition currently being evaluated, but also on all prior decisions as well. As a consequence, the structure that is synthesized takes into account this priority order, replacing the 4×2 AND-OR by a chain of four 2-to-1 multiplexers. In contrast, there is no direct dependency between the decisions made in with-select. With-select produces a decoder and the 4×2 AND-OR gate.

We have now covered many of the VHDL fundamentals needed for describing combinational circuits. We will continue with more on VHDL by presenting means for describing arithmetic circuits in Chapter 5 and sequential circuits in Chapter 6.

4-8 HDL REPRESENTATIONS FOR COMBINATIONAL CIRCUITS—VERILOG

Since an HDL is used for describing and designing hardware, it is very important to keep the underlying hardware in mind as you write in the language. This is particularly critical if your language description is to be synthesized. For example, if you ignore the hardware that will be generated, it is very easy to specify a large complex gate structure by using \times (multiply), when a much simpler structure using only a few gates is all that is needed. For this reason, initially, we emphasize describing detailed hardware with Verilog, and finishing with more abstract, higher-level descriptions.

Selected examples in this chapter are useful for introducing Verilog as an alternative means for representing detailed digital circuits. First, we show a structural Verilog description that replaces the schematic for the 2-to-4-line decoder with enable given in Figure 4-10 on page 152. This example, and one using the 4-to-1-line multiplexer in Figure 4-14 on page 158, illustrate many of the fundamental concepts of Verilog. We then present higher level functional and behavioral Verilog descriptions for these circuits that further illustrate Verilog concepts.

EXAMPLE 4-20 Structural Verilog for a 2-to-4-Line Decoder

The Verilog description for the 2-to-4-line decoder circuit from Figure 4-10 on page 152 is given in Figure 4-32. This description will be used to introduce a number of general Verilog features, as well as to illustrate structural circuit description.

The text between two slashes `//` and the end of a line as shown in lines 1 and 2 of Figure 4-32 is interpreted as a comment. For multiline comments, there is an alternative notation using a `/` and `*`:

```
/* 2-to-4 Line Decoder with Enable: Structural Verilog Desc.
(See Figure 4-10 for logic diagram)*/
```

```

// 2-to-4 Line Decoder with Enable: Structural Verilog Desc.      // 1
// (See Figure 4-10 for logic diagram)                          // 2
module decoder_2_to_4_st_v(EN, A0, A1, D0, D1, D2, D3);        // 3
    input EN, A0, A1;                                           // 4
    output D0, D1, D2, D3;                                       // 5
                                                                // 6
    wire A0_n, A1_n, N0, N1, N2, N3;                             // 7
    not                                                                // 8
        g0(A0_n, A0),                                           // 9
        g1(A1_n, A1);                                           // 10
    and                                                                // 11
        g3(N0, A0_n, A1_n),                                     // 12
        g4(N1, A0, A1_n),                                       // 13
        g5(N2, A0_n, A1),                                       // 14
        g6(N3, A0, A1),                                         // 15
        g7(D0, N0, EN),                                         // 16
        g8(D1, N1, EN),                                         // 17
        g9(D2, N2, EN),                                         // 18
        g10(D3, N3, EN);                                        // 19
endmodule                                                       // 20

```

□ FIGURE 4-32

Structural Verilog Description of 2-to-4-Line Decoder

To assist in discussion of the Verilog description, comments providing line numbers have been added on the right. As a language, Verilog has a syntax that describes precisely the valid constructs that can be used in the language. This example will illustrate many aspects of the syntax. In particular, note the use of commas and colons in the description. Commas (,) are typically used to separate elements of a list and semicolons (;) are used to terminate Verilog statements.

Line 3 begins the declaration of a **module**, which is the fundamental building block of a Verilog design. The remainder of the description defines the module, ending in line 20 with **endmodule**. Note that there is no ; after **endmodule**. Just as for a symbol in a schematic, we need to give the design a name and to define its inputs and outputs. This is the function of the *module statement* in line 3 and the *input* and *output declarations* that follow. The words **module**, **input** and **output** are keywords in Verilog. Keywords, which we show in bold type, have a special meaning and cannot be used as names of objects such as modules, inputs, outputs, or wires. The statement **module** decoder_2_to_4_st_v declares that a design or design part exists with the name decoder_2_to_4_st_v. Further, Verilog names are case sensitive (i.e., names are distinguished by the use of uppercase or lowercase letters). DECODER_2_4_st_v, Decoder_2_4_st_v, and decoder_2_4_st_V are all distinct names.

Just as we would do for a symbol in a schematic, we give the names of the decoder inputs and outputs in the module statement. Next, an *input declaration* is used to define which of the names in the module statement are inputs. For the example design, there

are three input signals EN, A0, and A1. The fact that these are inputs is denoted by the keyword **input**. Similarly, an *output declaration* is used to define the outputs. D0, D1, D2, and D3 are denoted as outputs by the keyword **output**.

Inputs and outputs as well as other binary signal types in Verilog can take on one of four values. The two obvious values are 0 and 1. Added are x to represent unknown values and z to represent high impedance values, on the outputs of 3-state logic. Verilog also has strength values that, when combined with the four values given, provide 120 possible signal states. Strength values are used in electronic circuit modeling, however, so will not be considered here.

STRUCTURAL DESCRIPTION Next, we want to specify the function of the decoder. In this case, we use a *structural description* that is equivalent to the circuit schematic given in Figure 4-10 on page 152. Note that the schematic is made up of gates. Verilog provides 14 primitive gates as keywords. Of these, we are interested in eight for now: **buf**, **not**, **and**, **or**, **nand**, **nor**, **xor** and **xnor**. **buf** and **not** have single inputs, and all other gate types may have from two to any integer number of inputs. **buf** is a buffer, which has the function $z = x$, with x as the input and z as the output. It is as an amplifier of electronic signals that can be used to provide greater fanout or smaller delays. **xor** is the exclusive-OR gate and **xnor** is the exclusive-NOR gate, the complement of the exclusive-OR. In our example, we will use just two gate types, **not** and **and** as shown in lines 8 and 11 of Figure 4-32.

Before specifying the interconnection of the gates, which is the same as a circuit netlist, we need to name all of the nets in the circuit. The inputs and outputs already have names. The internal nets are the outputs of the two inverters and of the four leftmost AND gates in Figure 4-10. In line 7, these nets are declared as *wires* by use of the keyword **wire**. Names A0_n and A1_n are used for the inverter outputs and N0, N1, N2, and N3 for the outputs of the AND gates. In Verilog, **wire** is the default net type. Notably, **input** and **output** ports have the default type **wire**.

Following the declaration of the internal signals, the circuit described contains two inverters and eight 2-input AND gates. A statement consists of a gate type followed by a list of instances of that gate type separated by commas. Each instance consists of a gate name and, enclosed in parentheses, the gate output and inputs separated by commas, with the output given first. The first statement begins on line 8 with the **not** gate type. Following is inverter g0 with A0_n as the output and A0 as the input. To complete the statement, g1 is similarly described. Lines 11 through 19 give the remaining eight gates and the signals connected to their outputs and inputs respectively. For example, in line 14, an instance of a 2-input AND gate named g5 is defined. It has output N2 and inputs A0_n and A1. The module is completed with the keyword **endmodule**. ■

EXAMPLE 4-21 Structural Verilog for a 4-to-1-Line Multiplexer

In Figure 4-33, the structural description of the 4-to-1-line multiplexer from Figure 4-14 on page 158 illustrates the Verilog concept of a vector. In lines 4 and 5, instead of specifying S and I as single bit wires, they are specified as multiple bit


```

// 4-to-1 Line Multiplexer: Structural Verilog Description           // 1
// (See Figure 4-14 for logic diagram)                             // 2
module multiplexer_4_to_1_st_v(S, I, Y);                           // 3
    input [1:0] S;                                                  // 4
    input [3:0] I;                                                  // 5
    output Y;                                                       // 6
                                                                    // 7
    wire [1:0] not_S;                                               // 8
    wire [0:3] D, N;                                               // 9
                                                                    // 10
not                                                                    // 11
    gn0(not_S[0], S[0]),                                           // 12
    gn1(not_S[1], S[1]);                                           // 13
                                                                    // 14
and                                                                    // 15
    g0(D[0], not_S[1], not_S[0]),                                   // 16
    g1(D[1], not_S[1], S[0]),                                       // 17
    g2(D[2], S[1], not_S[0]),                                       // 18
    g3(D[3], S[1], S[0]);                                           // 19
    g0(N[0], D[0], I[0]),                                           // 20
    g1(N[1], D[1], I[1]),                                           // 21
    g2(N[2], D[2], I[2]),                                           // 22
    g3(N[3], D[3], I[3]);                                           // 23
                                                                    // 24
or go(Y, N[0], N[1], N[2], N[3]);                                   // 25
                                                                    // 26
endmodule                                                           // 27

```

□ **FIGURE 4-33**
Structural Verilog Description of 4-to-1-Line Multiplexer

wires called *vectors*. The bits of a vector are named by a range of integers. This range is given by maximum and minimum values. By specifying these two values, we specify the width of the vector and the names of each of its bits. Vector ranges are illustrated in lines 4, 5, 8 and 9 of Figure 4-33. **input** [1:0] *S* indicates that *S* is a vector with a width of two, with the most significant bit numbered 1 and least significant bit numbered 0. The components of *S* are *S*[1] and *S*[0]. **input** [3:0] *I* declares *I* as a 4-bit input, with the most significant bit numbered 3 and least significant bit numbered 0. **wire** [0:3] *D* is also a 4-bit vector representing the four internal wires between the leftmost and rightmost AND gates, but in this case, the most significant bit is numbered 0 and the least significant bit is numbered 3. Once a vector has been declared, then the entire vector or its subcomponents can be referenced. For example, *S* refers to the two bits of *S*, and *S*[1] refers to the most significant bit of *S*. *N* refers to all four bits of *N* and *N*[1:2] refers to the middle two bits of *N*. These types of references are used in specifying the output and inputs in instances of the gates in lines 11 through 25. Otherwise, this Verilog description is similar in structure to that for the 2-to-4-line decoder, except that the schematic represented is that in Figure 4-14. ■

DATAFLOW DESCRIPTION A dataflow description is a form of Verilog description that is not based on structure, but rather on function. A dataflow description is made up of dataflow statements. For the first dataflow description, Boolean equations are used rather than the equivalent of a logic schematic. The Boolean equations given are executed in parallel whenever one of the values on the right-hand side of the equation changes.

EXAMPLE 4-22 Dataflow Verilog for a 2-to-4-Line Decoder

In Figure 4-34, a dataflow description is given for the 2-to-4-line decoder. This particular dataflow description uses the assignment statement consisting of the keyword **assign** followed, in this case, by a Boolean equation. In such equations, we use the bitwise Boolean operators given in Table 4-12. In line 7 of Figure 4-34, $\sim A0$ and $\sim A1$ are combined with an $\&$ operator. This $\&$ combination is assigned to the output D0. D1, D2, and D3 are similarly defined in lines 8 through 10. ■

In the next three examples, we describe the 4-to-1-line multiplexer to illustrate three alternative forms of data flow description: Boolean equations, binary combinations as conditions, and binary decisions as conditions.

EXAMPLE 4-23 Dataflow Verilog for a 4-to-1-Line Multiplexer

In Figure 4-35, a single Boolean equation for Y describes the multiplexer. This equation is in sum-of-products form with $\&$ for AND and $|$ for OR. Components of the S and I vectors are used as its variables. ■

EXAMPLE 4-24 Verilog for a 4-to-1-Line Multiplexer Using Combinations

The description in Figure 4-36 resembles the function table given as Table 4-7 on page 157 by using a conditional operator on binary combinations. If the logical value within the parentheses is true, then the value before the $:$ is assigned to the independent variable, in this case, Y. If the logical value is false, then the value after the $:$ is assigned. The logical equality operator is denoted by $==$. Suppose we consider condition $S == 2'b00$. $2'b00$ represents a constant. The 2 specifies that the constant contains two digits, b that the constant is given in binary, and 00 gives the constant value. Thus, the expression has value true if vector S is equal to 00;

□ **TABLE 4-12**
Bitwise Verilog Operators

Operation	Operator
\sim	Bitwise NOT
$\&$	Bitwise AND
$ $	Bitwise OR
\wedge	Bitwise XOR
$\wedge \sim$ or $\sim \wedge$	Bitwise XNOR

```

// 2-to-4 Line Decoder with Enable: Dataflow Verilog Desc.           // 1
// (See Figure 4-10 for logic diagram)                             // 2
module decoder_2_to_4_df_v(EN, A0, A1, D0, D1, D2, D3);           // 3
    input EN, A0, A1;                                             // 4
    output D0, D1, D2, D3;                                         // 5
                                                                    // 6
    assign D0 = EN & ~A1 & ~A0;                                     // 7
    assign D1 = EN & ~A1 & A0;                                     // 8
    assign D2 = EN & A1 & ~A0;                                     // 9
    assign D3 = EN & A1 & A0;                                     // 10
                                                                    // 11
endmodule                                                         // 12

```

□ FIGURE 4-34

Dataflow Verilog Description of 2-to-4-Line Decoder

```

// 4-to-1 Line Multiplexer: Dataflow Verilog Description
// (See Figure 4-14 for logic diagram)
module multiplexer_4_to_1_df_v(S, I, Y);
    input [1:0] S;
    input [3:0] I;
    output Y;

    assign Y = (~ S[1] & ~ S[0] & I[0]) | (~ S[1] & S[0] & I[1])
              | (S[1] & ~ S[0] & I[2]) | (S[1] & S[0] & I[3]);
endmodule

```

□ FIGURE 4-35

Dataflow Verilog Description of 4-to-1-Line Multiplexer Using a Boolean Equation

otherwise, it is false. If the expression is true, then $I[0]$ is assigned to Y . If the expression is false, then the next expression containing a $?$ is evaluated, and so on. In this case, for a condition to be evaluated, all conditions preceding it must evaluate to false. If none of the decisions evaluate to true, then the default value $1'bx$ is assigned to Y . Recall that default value x represents unknown. ■

EXAMPLE 4-25 Verilog for a 4-to-1-Line Multiplexer Using Binary Decisions

The final form of dataflow description is shown in Figure 4-37. It is based on conditional operators used to form a decision tree, which corresponds to a factored Boolean expression. In this case, if $S[1]$ is 1, then $S[0]$ is evaluated to determine whether Y is assigned $I[3]$ or assigned $I[2]$. If $S[1]$ is 0, then $S[0]$ is evaluated to determine whether Y is assigned $I[1]$ or $I[0]$. For a regular structure such as a multiplexer, this approach, based on two-way (binary) decisions, gives a simple dataflow expression. ■

```
// 4-to-1 Line Multiplexer: Dataflow Verilog Description
// (See Table 4-7 for function table)
module multiplexer_4_to_1_cf_v(S, I, Y);
    input [1:0] S;
    input [3:0] I;
    output Y;

    assign Y = (S == 2'b00) ? I[0] :
               (S == 2'b01) ? I[1] :
               (S == 2'b10) ? I[2] :
               (S == 2'b11) ? I[3] : 1'bx ;
endmodule
```

□ FIGURE 4-36

Conditional Dataflow Verilog Description of 4-to-1-Line Multiplexer Using Combinations

```
// 4-to-1 Line Multiplexer: Dataflow Verilog Description
// (See Table 4-7 for function table)
module multiplexer_4_to_1_tf_v(S, I, Y);
    input [1:0] S;
    input [3:0] I;
    output Y;

    assign Y = S[1] ? (S[0] ? I[3] : I[2]) :
               (S[0] ? I[1] : I[0]) ;
endmodule
```

□ FIGURE 4-37

Conditional Dataflow Verilog Description of 4-to-1-Line Multiplexer Using Binary Decisions

This completes our initial introduction to Verilog. We will continue with more on Verilog by presenting means for describing arithmetic circuits in Chapter 5 and sequential circuits in Chapter 6.

4-9 CHAPTER SUMMARY

This chapter has dealt with a number of combinational circuit types, called functional blocks, that are frequently used to design larger circuits. Rudimentary circuits that implement functions of a single variable were introduced. The design of decoders that activate one of a number of output lines in response to an input code was covered. Encoders, the inverse of decoders, generate a code associated with the active line from a set of lines. The design of multiplexers, which take data applied at the input selected and present it at the output, was illustrated.

The design of combinational logic circuits using decoders, multiplexers, and programmable logic was covered. In combination with OR gates, decoders

provide a simple minterm-based approach to implementing combinational circuits. Procedures were given for using an n -to-1-line multiplexer or a single inverter and an $(n - 1)$ -to-1-line multiplexer to implement any n -input Boolean function. ROMs can be programmed with truth tables. PLAs and PALs can be programmed with their own specialized programming tables. Multiple-level logic decomposition and extraction from Chapter 2 map combinational equations for lookup table implementations.

The last two sections of the chapter introduced VHDL and Verilog descriptions for combinational circuits. Each of the HDLs was illustrated by descriptions at structural, functional, and behavioral levels for various functional blocks presented earlier in the chapter.

REFERENCES

1. MANO, M. M. *Digital Design*, 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 2002.
2. WAKERLY, J. F. *Digital Design: Principles and Practices*, 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 2000.
3. *High-Speed CMOS Logic Data Book*. Dallas: Texas Instruments, 1989.
4. *IEEE Standard VHDL Language Reference Manual*. (ANSI/IEEE Std 1076-1993; revision of IEEE Std 1076-1987). New York: The Institute of Electrical and Electronics Engineers, 1994.
5. SMITH, D. J. *HDL Chip Design*. Madison, AL: Doone Publications, 1996.
6. PELLERIN, D. AND D. TAYLOR. *VHDL Made Easy!* Upper Saddle River, NJ: Prentice Hall PTR, 1997.
7. STEFAN, S. AND L. LINDH. *VHDL for Designers*. London: Prentice Hall Europe, 1997.
8. YALAMANCHILI, S. *VHDL Starter's Guide*. Upper Saddle River, NJ: Prentice Hall, 1998.
9. *IEEE Standard Description Language Based on the Verilog(TM) Hardware Description Language* (IEEE Std 1364-1995). New York: The Institute of Electrical and Electronics Engineers, 1995.
10. PALNITKAR, S. *Verilog HDL: A Guide to Digital Design and Synthesis*. Upper Saddle River, NJ: SunSoft Press (A Prentice Hall Title), 1996.
11. BHASKER, J. A *Verilog HDL Primer*. Allentown, PA: Star Galaxy Press, 1997.
12. THOMAS, D., AND P. MOORBY. *The Verilog Hardware Description Language* 4th ed. Boston: Kluwer Academic Publishers, 1998.
13. CILETTI, M. *Modeling, Synthesis, and Rapid Prototyping with the Verilog HDL*, Upper Saddle River, NJ: Prentice Hall, 1999.

PROBLEMS



The plus (+) indicates a more advanced problem and the asterisk (*) indicates a solution is available on the text website.

- 4-1. *** (a) Draw an implementation diagram for a constant vector function $F = (F_7, F_6, F_5, F_4, F_3, F_2, F_1, F_0) = (1, 0, 0, 1, 0, 1, 1, 0)$ using the ground and power symbols from Figure 4-2(b).
- (b) Draw an implementation diagram for a rudimentary vector function $G = (G_7, G_6, G_5, G_4, G_3, G_2, G_1, G_0) = (A, \bar{A}, 0, 1, \bar{A}, A, 1, 1)$ using inputs 1, 0, A , and \bar{A} .
- 4-2.** (a) Draw an implementation diagram for rudimentary vector function $F = (F_7, F_6, F_5, F_4, F_3, F_2, F_1, F_0) = (1, 0, A, \bar{A}, \bar{A}, A, 0, 1)$, using the ground and power symbols in Figure 4-2(b) and the wire and inverter in Figure 4-2(c) and 4-2(d).
- (b) Draw an implementation diagram for rudimentary vector function $G = (G_7, G_6, G_5, G_4, G_3, G_2, G_1, G_0) = (F_3, \bar{F}_2, 0, 1, \bar{F}_1, F_0, 1, 1)$, using the ground and power symbols and components of vector F .
- 4-3.** (a) Draw an implementation diagram for the vector $G = (G_7, G_6, G_5, G_4, G_3, G_2, G_1, G_0) = (F_{11}, F_{10}, F_9, F_8, F_3, F_2, F_1, F_0)$.
- (b) Draw a simple implementation for the rudimentary vector $H = (H_7, H_6, H_5, H_4, H_3, H_2, H_1, H_0) = (F_3, F_2, F_1, F_0, G_3, G_2, G_1, G_0)$.
- 4-4.** A home security system has a master switch that is used to enable an alarm, lights, video cameras, and a call to local police in the event one or more of six sets of sensors detects an intrusion. The inputs, outputs, and operation of the enabling logic are specified as follows:
- Inputs:**
- $S_i, i = 0, 1, 2, 3, 4, 5$ - signals from six sensor sets (0 - intrusion detected, 1 - no intrusion detected)
 - M - master switch (0 - security system on, 1 - security system off)
- Outputs:**
- A - alarm (0 - alarm on, 1 - alarm off)
 - L - lights (0 - lights on, 1 - lights off)
 - V - video cameras (0 - video cameras off, 1 - video cameras on)
 - C - call to police (0 - call off, 1 - call on)
- Operation:**
- If one or more of the sets of sensors detects an intrusion and the security system is on, then all outputs are on. Otherwise, all outputs are off.
- Find a minimum gate input count realization of the enabling logic using AND and OR gates and inverters.
- 4-5.** Design a 4-to-16-line decoder using two 3-to-8-line decoders and 16 2-input AND gates.
- 4-6.** Design a 4-to-16-line decoder with enable using five 2-to-4-line decoders with enables as shown in Figure 4-10.

- 4-7. *Design a 5-to-32-line decoder using four 3-to-8-line decoders and 48 2-input AND gates.
- 4-8. A special 4-to-6-line decoder is to be designed. The input codes used are 000 through 101. For a given code applied, the output D_i , with i equal to the decimal equivalent of the code, is 1 and all other outputs are 0. Design the decoder with a 2-to-4-line decoder, a 1-to-2-line decoder, and six 2-input AND gates, such that all decoder outputs are used at least once.
- 4-9. Draw the detailed logic diagram of a 3-to-8-line decoder using only NOR and NOT gates. Include an enable input.
- 4-10. *Design a 4-input priority encoder with inputs and outputs as in Table 4-5, but with the truth table representing the case in which input D_0 has the highest priority and input D_3 has the lowest priority.
- 4-11. Derive the truth table of a BCD-to-binary priority encoder.
- 4-12. (a) Design an 8-to-1-line multiplexer using a 3-to-8-line decoder and an 8×2 AND-OR.
 (b) Repeat part (a), using two 4-to-1-line multiplexers and one 2-to-1-line multiplexer.
- 4-13. Design a 16-to-1-line multiplexer using a 4-to-16-line decoder and a 16×2 AND-OR.
- 4-14. Design a dual 8-to-1-line decoder using a 3-to-8-line decoder and two 8×2 AND-ORs.
- 4-15. Design a dual 4-to-1-line multiplexer using a 2-to-4-line decoder and eight 3-state buffers.
- 4-16. Design an 8-to-1-line multiplexer using transmission gates.
- 4-17. Construct a 10-to-1-line multiplexer with a 3-to-8-line decoder, a 1-to-2-line decoder, and a 10×3 AND-OR. The selection codes 0000 through 1001 must be directly applied to the decoder inputs without added logic.
- 4-18. Construct a quad 9-to-1-line multiplexer with four single 8-to-1-line multiplexers and one quadruple 2-to-1-line multiplexer. The multiplexers should be interconnected and inputs labeled so that the selection codes 0000 through 1000 can be directly applied to the multiplexer selection inputs without added logic.
- 4-19. *Construct a 15-to-1-line multiplexer with two 8-to-1-line multiplexers. Interconnect the two multiplexers and label the inputs such that any added logic required to have selection codes 0000 through 1110 is minimized.
- 4-20. Rearrange the condensed truth table for the circuit of Figure 4-10, and verify that the circuit can function as a demultiplexer.

4-21. A combinational circuit is defined by the following three Boolean functions:

$$F_1 = \overline{X + Z} + XYZ$$

$$F_2 = \overline{X + Z} + \overline{X}YZ$$

$$F_3 = X\overline{Y}Z + \overline{X + Z}$$

Design the circuit with a decoder and external OR gates.

4-22. A combinational circuit is specified by the following three Boolean functions:

$$F_1(A, B, C) = \Sigma m(0, 3, 4)$$

$$F_2(A, B, C) = \Sigma m(1, 2, 7)$$

$$F_3(A, B, C) = \Pi M(0, 1, 2, 4)$$

Implement the circuit with a decoder and external OR gates.

4-23. Implement a binary full adder with a dual 4-to-1-line multiplexer and a single inverter.

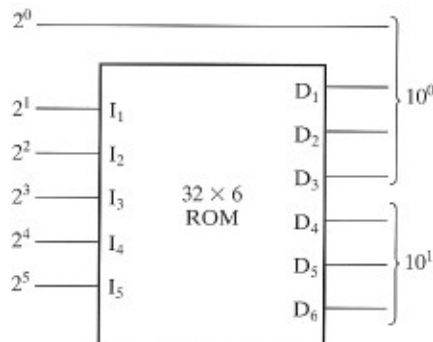
4-24. Implement the following Boolean function with an 8-to-1-line multiplexer and a single inverter with variable D as its input:

$$F(A, B, C, D) = \Sigma m(2, 4, 6, 9, 10, 11, 15)$$

4-25. *Implement the Boolean function

$$F(A, B, C, D) = \Sigma m(1, 3, 4, 11, 12, 13, 14, 15)$$

with a 4-to-1-line multiplexer and external gates. Connect inputs A and B to the selection lines. The input requirements for the four data lines will be a



□ **FIGURE 4-38**
Binary-to-Decimal ROM Converter

function of the variables C and D . The values of these variables are obtained by expressing F as a function of C and D for each of the four cases when $AB = 00, 01, 10$, and 11 . These functions must be implemented with external gates.

- 4-26.** Repeat problem 4-25 using two 3-to-8-line decoders with enables, an inverter, and OR gates with a maximum fan-in of 4.
- 4-27.** Given a 256×8 ROM chip with an enable input, show the external connections necessary to construct a $1K \times 16$ ROM with eight chips and a decoder.
- 4-28.** *The 32×6 ROM, together with the 2^0 line, as shown in Figure 4-38, converts a 6-bit binary number to its corresponding two-digit BCD number. For example, binary 100001 converts to BCD 011 0011 (decimal 33). Specify the truth table for the ROM.
- 4-29.** Specify the size of a ROM (number of words and number of bits per word) that will accommodate the truth table for the following combinational circuit components:
- (a) An 8-bit adder-subtractor with C_{in} and C_{out} .
 - (b) A binary multiplier that multiplies two 8-bit numbers.
 - (c) A code converter from a 4-digit BCD number to a binary number.
- 4-30.** Tabulate the truth table for an 8×4 ROM that implements the following four Boolean functions:

$$A(X, Y, Z) = \Sigma m(0, 1, 2, 6, 7)$$

$$B(X, Y, Z) = \Sigma m(2, 3, 4, 5, 6)$$

$$C(X, Y, Z) = \Sigma m(2, 6)$$

$$D(X, Y, Z) = \Sigma m(1, 2, 3, 5, 6, 7)$$

- 4-31.** Obtain the PLA programming table for the four Boolean functions listed in Problem 4-30. Minimize the number of product terms. Be sure to attempt to share product terms between functions that are not prime implicants of individual functions and to consider the use of complemented (C) outputs.
- 4-32.** Derive the PLA programming table for the combinational circuit that squares a 3-bit number. Minimize the number of product terms.
- 4-33.** List the PLA programming table for a BCD-to-Excess-3 code converter.
- 4-34.** *Repeat Problem 4-33, using a PAL device.
- 4-35.** The following is the truth table of a three-input, four-output combinational circuit. Obtain the PAL programming table for the circuit, and mark the fuses to be blown in a PAL diagram similar to the one shown in Figure 4-26.

Inputs			Outputs			
X	Y	Z	A	B	C	D
0	0	0	0	1	0	0
0	0	1	1	1	1	1
0	1	0	1	0	1	1
0	1	1	0	1	0	1
1	0	0	1	0	1	0
1	0	1	0	0	0	1
1	1	0	1	1	1	0
1	1	1	0	1	1	1



All HDL files for circuits referred to in the remaining problems are available in ASCII form for simulation and editing on the Companion Website for the text. A VHDL or Verilog compiler/simulator is necessary for the problems or portions of problems requesting simulation. Descriptions can still be written, however, for many problems without using compilation or simulation.

- 4-36. Compile and simulate the 2-to-4-line decoder with enable in Figure 4-27 for sequence 000, 001, 010, 011, 100, 101, 110, 111 on E_n, A0, A1. Verify that the circuit functions as a decoder. You will need to compile library `lcdf_vhdl.func_prims` first since it is used in the simulation.
- 4-37. Rewrite the VHDL given in Figure 4-27 for the 2-to-4-line decoder using (1) `std_logic_vector` notation instead of `std_logic` notation for A and D_n and (2) implicit specification of the component input and output names by their order in package `func_prims` in library `lcdf_vhdl` given in the Companion Website Gallery. See Figure 4-28 and accompanying text for these concepts. Compile and simulate the resulting file as in problem 4-36.
- 4-38. Compile and simulate the 4-to-1-line multiplexer in Figure 4-28 for the sequence of all 16 combinations of 00, 10, 01, 11 on S and 1000, 0100, 0010, 0001 on D. You will need to compile library `lcdf_vhdl.func_prims` first since it is used in the simulation. Verify that the circuit functions as a multiplexer.
- 4-39. *Find a logic diagram that corresponds to the VHDL structural description in Figure 4-39. Note that complemented inputs are not available.
- 4-40. Using Figure 4-28 as a framework, write a structural VHDL description of the circuit in Figure 4-40. Replace X, Y, and Z with `X(0:2)`. Consult package `func_prims` in library `lcdf_vhdl` for information on the various gate components. Compile `func_prims` and your VHDL, and simulate your VHDL for all eight possible input combinations to verify your description's correctness.
- 4-41. Using Figure 4-27 as a framework, write a structural VHDL description of the circuit in Figure 4-41. Consult package `func_prims` in library

```

-- Combinational Circuit 1: Structural VHDL Description
library ieee, lcdf_vhdl;
use ieee.std_logic_1164.all, lcdf_vhdl.func_prims.all;
entity comb_ckt_1 is
    port(x1, x2, x3, x4 : in std_logic;
         f : out std_logic);
end comb_ckt_1;

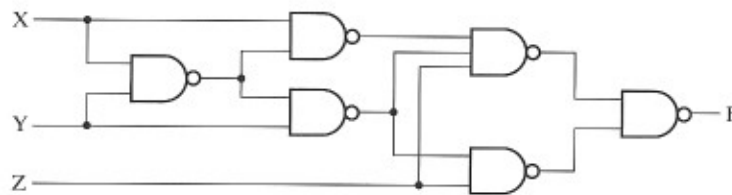
architecture structural_1 of comb_ckt_1 is
    component NOT1
        port(in1: in std_logic;
             out1: out std_logic);
    end component;
    component AND2
        port(in1, in2 : in std_logic;
             out1: out std_logic);
    end component;
    component OR3
        port(in1, in2, in3 : in std_logic;
             out1: out std_logic);
    end component;
    signal n1, n2, n3, n4, n5, n6 : std_logic;
begin
    g0: NOT1 port map (in1 => x1, out1 => n1);
    g1: NOT1 port map (in1 => n3, out1 => n4);
    g2: AND2 port map (in1 => x2, in2 => n1,
                      out1 => n2);
    g3: AND2 port map (in1 => x2, in2 => x3,
                      out1 => n3);
    g4: AND2 port map (in1 => x3, in2 => x4,
                      out1 => n5);
    g5: AND2 port map (in1 => x1, in2 => n4,
                      out1 => n6);
    g6: OR3 port map (in1 => n2, in2 => n5,
                     in3 => n6, out1 => f);
end structural_1;

```

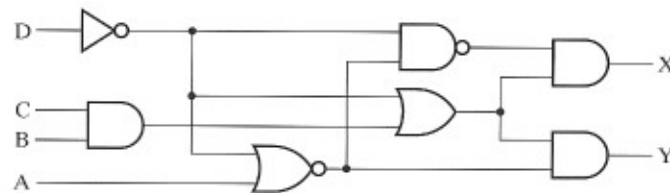
□ **FIGURE 4-39**
VHDL for Problem 4-39

lcdf_vhdl for information on the various gate components. Compile func_prims and your VHDL, and simulate your VHDL for all 16 possible input combinations to verify your description's correctness.

- 4-42.** Find a logic diagram representing minimum two-level logic needed to implement the VHDL dataflow description in Figure 4-42. Note that complemented inputs are available.



□ **FIGURE 4-40**
Circuit for Problems 4-40, 4-43, 4-51, and 4-53



□ **FIGURE 4-41**
Circuit for Problem 4-41 and 4-50

```
-- Combinational Circuit 2: Dataflow VHDL Description
-library ieee;
use ieee.std_logic_1164.all;
entity comb_ckt_2 is
    port(a, b, c, d, a_n, b_n, c_n, d_n: in std_logic;
          f, g : out std_logic);
    -- a_n, b_n, ... are complements of a, b, ... , respectively.
end comb_ckt_2;

architecture dataflow_1 of comb_ckt_2 is
begin
    f <= b and (a or (a_n and c)) or (b_n and c and d_n);
    g <= b and (c or (a_n and c_n) or (c_n and d_n));
end dataflow_1;
```

□ **FIGURE 4-42**
VHDL for Problem 4-42

- 4-43.** *Write a dataflow VHDL description for the circuit in Figure 4-40 by using the Boolean equation for the output F .
- 4-44.** +Write a dataflow VHDL description for the priority encoder using the “when else” dataflow concept from Figure 4-30. Compile and simulate your

description with a set of input vectors that are a good test for the priority function it performs.

- 4-45. Write a dataflow VHDL description for an 8-to-1-line multiplexer using the “with select” dataflow concept from Figure 4-31. Compile and simulate your description with a set of input vectors that are a good test for the selection function it performs.
- 4-46. *Compile and simulate the 2-to-4-line decoder in Figure 4-32 for sequence 000, 001, 010, 011, 100, 101, 110, 111 on E, A0, A1. Verify that the circuit functions as a decoder.
- 4-47. Rewrite the Verilog description given in Figure 4-32 for the 2-to-4-line decoder using vector notation for inputs, outputs, and wires. See Figure 4-33 and accompanying text for these concepts. Compile and simulate the resulting file as in problem 4-46.
- 4-48. Compile and simulate the 4-to-1-line multiplexer in Figure 4-33 for the sequence of all 16 combinations of 00, 10, 01, 11 on S and 1000, 0100, 0010, 0001 on D. Verify that the circuit functions as a multiplexer.
- 4-49. *Find a logic diagram that corresponds to the Verilog structural description in Figure 4-43. Note that complemented inputs are not available.
- 4-50. Using Figure 4-32 as a framework, write a structural Verilog description of the circuit in Figure 4-41. Compile and simulate your Verilog for all 16 possible input combinations to verify your description’s correctness.
- 4-51. Using Figure 4-33 as a framework, write a structural Verilog description of the circuit in Figure 4-40. Replace X, Y, and Z with **input** [2:0] X. Compile

```
// Combinational Circuit 1: Structural Verilog Description
module comb_ckt_1(x1, x2, x3, x4,f);
    input x1, x2, x3, x4;
    output f;

    wire n1, n2, n3, n4, n5, n6;
    not
        go(n1, x1),
        g1(n4, n3);
    and
        g2(n2, x2, n1),
        g3(n3, x2, x3),
        g4(n5, x3, x4),
        g5(n6, x1, n4);
    or
        g6(f, n2, n5, n6),
endmodule
```

□ **FIGURE 4-43**
Verilog for Problem 4-49

```
// Combinational Circuit 2: Dataflow Verilog Description
module comb_ckt_1 (a, b, c, d, a_n, b_n, c_n, d_n, f, g);
// a_n, b_n, ... are complements of a, b, ... , respectively.
  input a, b, c, d, a_n, b_n, c_n, d_n;
  output f, g;

  assign f = b & (a | (a_n & c)) | (b_n & c & d_n);
  assign g = b & (c | (a_n & c_n) | (c_n & d_n));
endmodule
```

□ **FIGURE 4-44**
Verilog for Problem 4-52

and simulate your Verilog for all eight possible input combinations to verify your description's correctness.

- 4-52.** Find a logic diagram representing minimum 2-level logic needed to implement the Verilog dataflow description in Figure 4-44. Note that complemented inputs are available.
- 4-53.** *Write a dataflow Verilog description for the circuit in Figure 4-40 by using the Boolean equation for the output F and using Figure 4-35 as a model.
- 4-54.** By using the conditional dataflow concept from Figure 4-36, write a Verilog dataflow description for an 8-to-1-line multiplexer. Compile and simulate your description with a set of input vectors that are a good test for the selection function it performs.
- 4-55.** +Write a dataflow description for the priority encoder in Figure 4-12 using the binary decision dataflow concept from Figure 4-37. Compile and simulate your description with a set of input vectors that are a good test for the priority function it performs.

CHAPTER

5

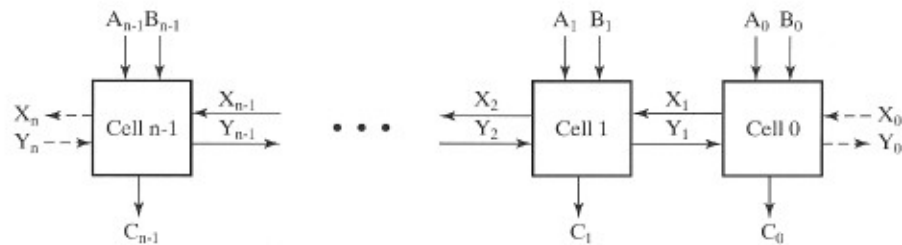
ARITHMETIC FUNCTIONS AND CIRCUITS

In this chapter, the focus continues to be on functional blocks, specifically, a special class of functional blocks that perform arithmetic operations. The concept of iterative circuits made up of arrays of combinational cells is introduced. Blocks designed as iterative arrays for performing addition, addition and subtraction, and multiplication are covered. The simplicity of these arithmetic circuits comes from using complement representations for numbers and complement-based arithmetic. In addition, we introduce circuit contraction that permits us to design new functional blocks. Contraction involves application of value-fixing to the inputs of existing blocks and simplification of the resulting circuits. These circuits perform operations such as incrementing a number, decrementing a number, or multiplying a number by a constant. Many of these new functional blocks are used to construct sequential functional blocks in Chapter 7.

In the generic computer diagram at the beginning of Chapter 1, adders, adder-subtractors, and multipliers are used in the processor. Incrementers and decrementers are used widely in other components as well, so concepts from this chapter apply across most components of the generic computer.

5-1 ITERATIVE COMBINATIONAL CIRCUITS

In this chapter, the arithmetic blocks are typically designed to operate on binary input vectors and produce binary output vectors. Further, the function implemented often requires that the same subfunction be applied to each bit position. Thus, a functional block can be designed for the subfunction and then used repetitively for each bit positions of the overall arithmetic block being designed. There



□ **FIGURE 5-1**
Block Diagram of an Iterative Circuit

will often be one or more connections to pass values between adjacent bit positions. These internal variables are inputs or outputs of the subfunctions, but not accessible outside of the overall arithmetic block. The subfunction blocks are referred to as *cells* and the overall implementation is an *array of cells*. The cells in the array are often, but not always, identical. Due to the repetitive nature of the circuit and the association of a vector index with each of the circuit cells, the overall functional block is referred to as an *iterative array*. The use of iterative arrays, a special case of hierarchical circuits, is useful in handling vectors of bits, for example, a circuit that adds two 32-bit binary integers. At a minimum, such a circuit has 64 inputs and 32 outputs. As a consequence, beginning with truth tables and writing equations for the entire circuit is out of the question. Since iterative circuits are based on repetitive cells, the design process is considerably simplified by a basic structure that guides the design.

A block diagram for an iterative circuit that operates on two n -input vectors and produces an n -output vector is shown in Figure 5-1. In this case, there are two lateral connections between each pair of cells in the array, one from left to right and the other from right to left. Also, optional connections, indicated by dashed lines, exist at the right and left ends of the array. An arbitrary array employs as many lateral connections as needed for a particular design. The definition of the functions associated with such connections is very important in the design of the array and its cell. In particular, the number of connections used and their functions can affect both the cost and speed of an iterative circuit.

In the next section, we will define cells for performing addition in individual bit positions and then define a binary adder as an array of cells.

5-2 BINARY ADDERS

An arithmetic circuit is a combinational circuit that performs arithmetic operations such as addition, subtraction, multiplication, and division with binary numbers or with decimal numbers in a binary code. We will develop arithmetic circuits by means of hierarchical, iterative design. We begin at the lowest level by finding a circuit that performs the addition of two binary digits. This simple addition consists of four possible elementary operations: $0 + 0 = 0$, $0 + 1 = 1$, $1 + 0 = 1$, and $1 + 1 = 10$. The first three operations produce a sum requiring only one bit to represent it, but

□ **TABLE 5-1**
Truth Table of Half Adder

Inputs		Outputs	
X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

when both the augend and addend are equal to 1, the binary sum requires two bits. Because of this case, the result is always represented by two bits, the carry and the sum. The carry obtained from the addition of two bits is added to the next higher order pair of significant bits. A combinational circuit that performs the addition of two bits is called a *half adder*. One that performs the addition of three bits (two significant bits and a previous carry) is called a *full adder*. The names of the circuits stem from the fact that two half adders can be employed to implement a full adder. The half adder and the full adder are basic arithmetic blocks with which other arithmetic circuits are designed.

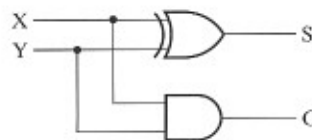
Half Adder

A half adder is an arithmetic circuit that generates the sum of two binary digits. The circuit has two inputs and two outputs. The input variables are the augend and addend bits to be added, and the output variables produce the sum and carry. We assign the symbols X and Y to the two inputs and S (for “sum”) and C (for “carry”) to the outputs. The truth table for the half adder is listed in Table 5-1. The C output is 1 only when both inputs are 1. The S output represents the least significant bit of the sum. The Boolean functions for the two outputs, easily obtained from the truth table, are

$$S = \overline{X}Y + X\overline{Y} = X \oplus Y$$

$$C = XY$$

The half adder can be implemented with one exclusive-OR gate and one AND gate, as shown in Figure 5-2.



□ **FIGURE 5-2**
 Logic Diagram of Half Adder

□ **TABLE 5-2**
Truth Table of Full Adder

Inputs			Outputs	
X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Full Adder

A full adder is a combinational circuit that forms the arithmetic sum of three input bits. Besides the three inputs, it has two outputs. Two of the input variables, denoted by X and Y , represent the two significant bits to be added. The third input, Z , represents the carry from the previous lower significant position. Two outputs are necessary because the arithmetic sum of three bits ranges in value from 0 to 3, and binary 2 and 3 need two digits for their representation. Again, the two outputs are designated by the symbols S for “sum” and C for “carry”; the binary variable S gives the value of the bit of the sum, and the binary variable C gives the output carry. The truth table of the full adder is listed in Table 5-2. The values for the outputs are determined from the arithmetic sum of the three input bits. When all the input bits are 0, the outputs are 0. The S output is equal to 1 when only one input is equal to 1 or when all three inputs are equal to 1. The C output has a carry of 1 if two or three inputs are equal to 1. The maps for the two outputs of the full adder are shown in Figure 5-3. The simplified sum-of-product functions for the two outputs are

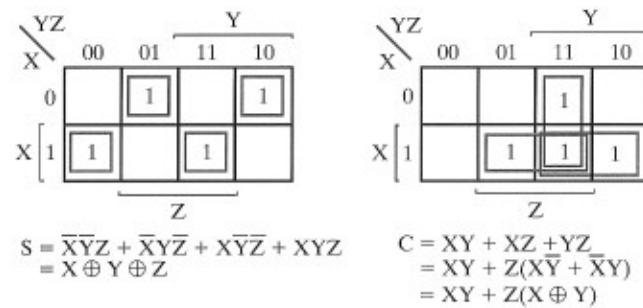
$$S = \bar{X}\bar{Y}Z + \bar{X}Y\bar{Z} + X\bar{Y}\bar{Z} + XYZ$$

$$C = XY + XZ + YZ$$

The two-level implementation requires seven AND gates and two OR gates. However, the map for output S is recognized as an odd function, as discussed in Section 2-7. Furthermore, the C output function can be manipulated to include the exclusive-OR of X and Y . The Boolean functions for the full adder in terms of exclusive-OR operations can then be expressed as

$$S = (X \oplus Y) \oplus Z$$

$$C = XY + Z(X \oplus Y)$$

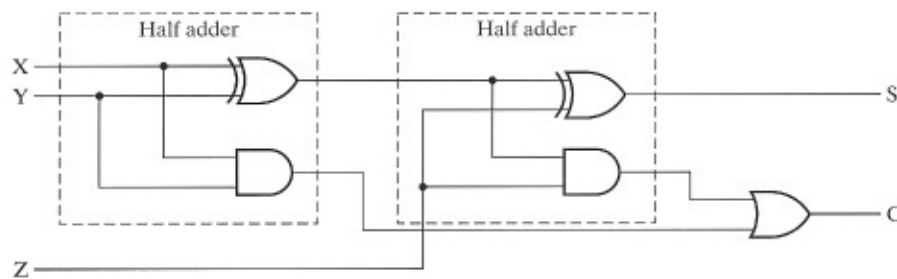


□ **FIGURE 5-3**
Maps for Full Adder

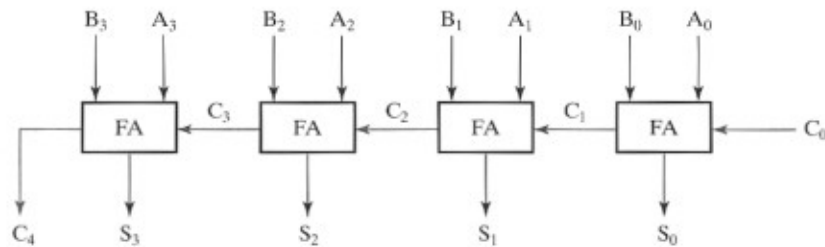
The logic diagram for this multiple-level implementation is shown in Figure 5-4. It consists of two half adders and an OR gate.

Binary Ripple Carry Adder

A parallel binary adder is a digital circuit that produces the arithmetic sum of two binary numbers using only combinational logic. The parallel adder uses n full adders in parallel, with all input bits applied simultaneously to produce the sum. The full adders are connected in cascade, with the carry output from one full adder connected to the carry input of the next full adder. Since a 1 carry may appear near the least significant bit of the adder and yet propagate through many full adders to the most significant bit, just as a wave ripples outward from a pebble dropped in a pond, the parallel adder is referred to as a *ripple carry adder*. Figure 5-5 shows the interconnection of four full-adder blocks to form a 4-bit ripple carry adder. The augend bits of A and the addend bits of B are designated by subscripts in increasing order from right to left, with subscript 0 denoting the least significant bit. The carries are connected in a chain through the full adders. The input carry to the parallel adder is C_0 , and the output carry is C_4 . An n -bit ripple carry adder requires n full adders, with each output carry connected to the input carry of the next-higher-order

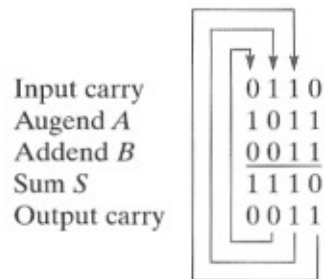


□ **FIGURE 5-4**
Logic Diagram of Full Adder



□ **FIGURE 5-5**
4-Bit Ripple Carry Adder

full adder. For example, consider the two binary numbers $A = 1011$ and $B = 0011$. Their sum, $S = 1110$, is formed with a 4-bit ripple carry adder as follows:



The input carry in the least significant position is 0. Each full adder receives the corresponding bits of A and B and the input carry and generates the sum bit for S and the output carry. The output carry in each position is the input carry of the next-higher-order position, as indicated by the blue lines.

The 4-bit adder is a typical example of a digital component that can be used as a building block. It can be used in many applications involving arithmetic operations. Observe that the design of this circuit by the usual method would require a truth table with 512 entries, since there are nine inputs to the circuit. By cascading the four instances of the known full adders, it is possible to obtain a simple and straightforward implementation without directly solving this larger problem. This is an example of the power of iterative circuits and circuit reuse in design.

Carry Lookahead Adder

The ripple carry adder, although simple in concept, has a long circuit delay due to the many gates in the carry path from the least significant bit to the most significant bit. For a typical design, the longest delay path through an n -bit ripple carry adder is $2n + 2$ gate delays. Thus, for a 16-bit ripple carry adder, the delay is 34 gate delays. This delay tends to be one of the largest in a typical computer design. Accordingly, we find an alternative design, the *carry lookahead adder*, attractive. This adder is a practical design with reduced delay at the price of more complex

hardware. The carry lookahead design can be obtained by a transformation of the ripple carry design in which the carry logic over fixed groups of bits of the adder is reduced to two-level logic. The transformation is shown for a 4-bit adder group in Figure 5-6.

First, we construct a new logic hierarchy, separating the parts of the full adders not involving the carry propagation path from those containing the path. We call the first part of each full adder a *partial full adder* (PFA). This separation is shown in Figure 5-6(a), which presents a diagram of a PFA and a diagram of four PFAs connected to the carry path. We have removed the OR gate and one of the AND gates from each of the full adders to form the ripple carry path.

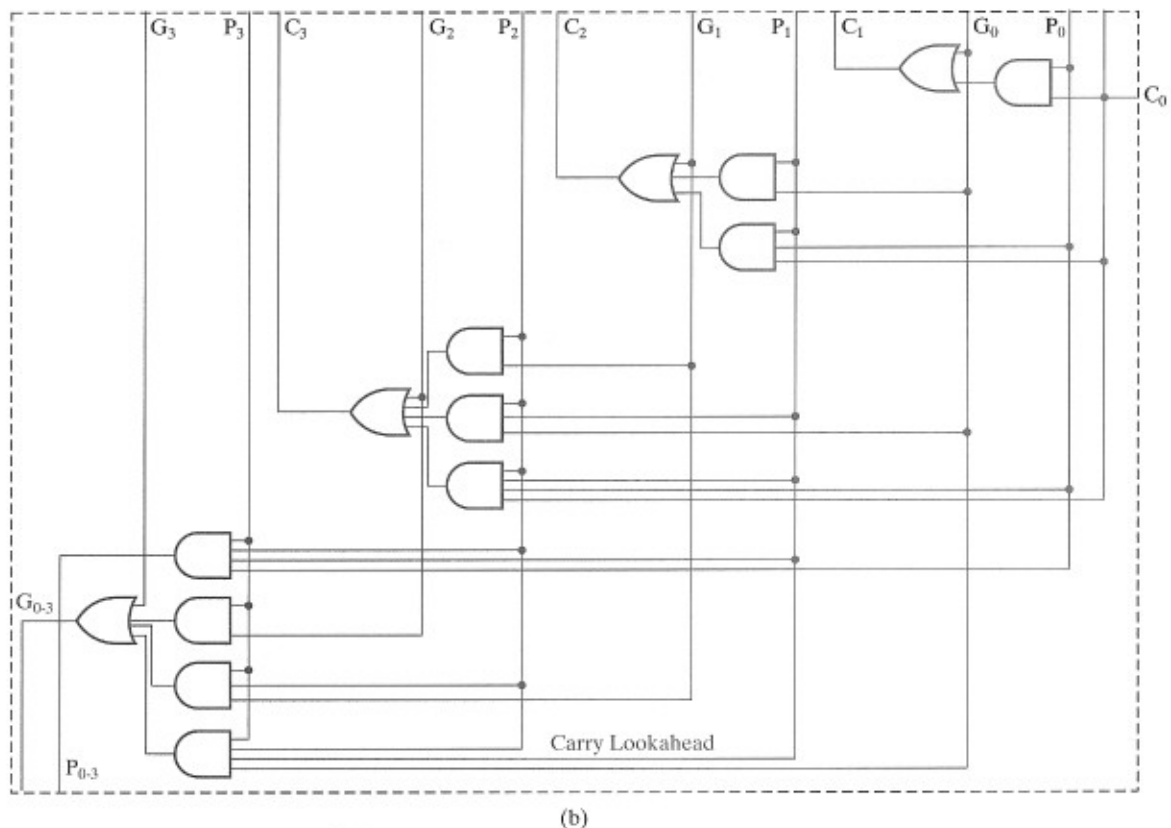
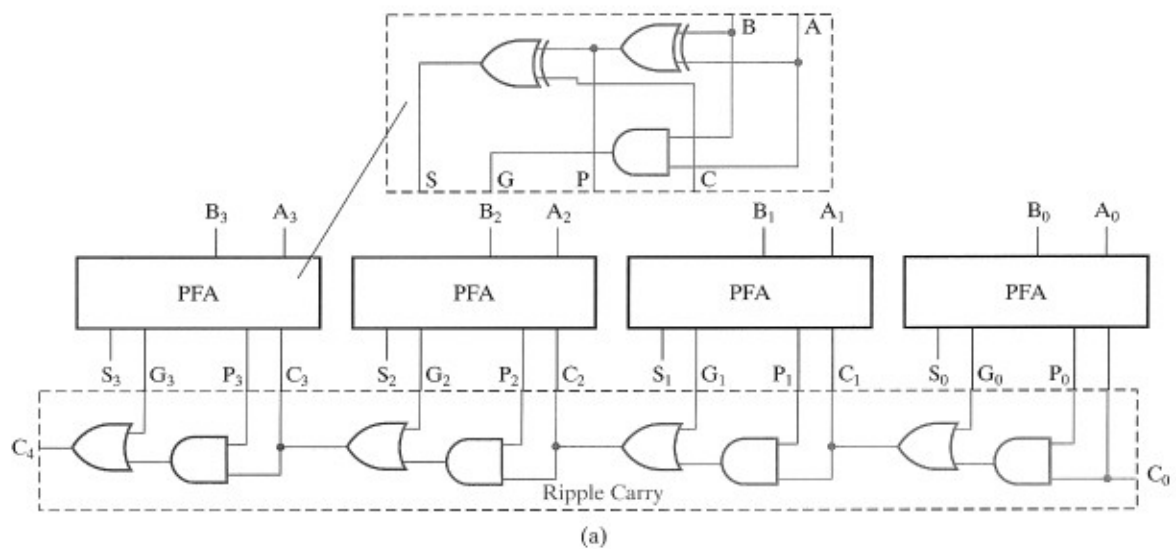
There are two outputs, P_i and G_i , from each PFA to the ripple carry path and one input C_i , the carry input, from the carry path to each PFA. The function $P_i = A_i \oplus B_i$ is called the *propagate* function. Whenever P_i is equal to 1, an incoming carry is propagated through the bit position from C_i to C_{i+1} . For P_i equal to 0, carry propagation through the bit position is blocked. The function $G_i = A_i \cdot B_i$ and is called the *generate* function. Whenever G_i is equal to 1, the carry output from the position is 1, regardless of the value of P_i , so a carry has been generated in the position. When G_i is 0, a carry is not generated, so that C_{i+1} is 0 if the carry propagated through the position from C_i is also 0. The generate and propagate functions correspond exactly to the half adder and are essential in controlling the values in the ripple carry path. Also, as in the full adder, the PFA generates the sum function by the exclusive-OR of the incoming carry C_i and the propagate function P_i .

The carry path remaining in the 4-bit ripple carry adder has a total of eight gates in cascade, so the circuit has a delay of eight gate delays. Since only AND and OR gates are involved in the carry path, ideally, the delay for each of the four carry signals produced, C_1 through C_4 , would be just two gate delays. The basic carry lookahead circuit is simply a circuit in which functions C_1 through C_3 have a delay of only two gate delays. The implementation of C_4 is more complicated in order to allow the 4-bit carry lookahead adder to be extended to multiples of 4 bits, such as 16 bits. The 4-bit carry lookahead circuit is shown in Figure 5-6(b). It is designed to directly replace the ripple carry path in Figure 5-6(a). Since the logic generating C_1 is already two-level, it remains unchanged. The logic for C_2 , however, has four levels. So to find the carry lookahead logic for C_2 , we must reduce the logic to two levels. The equation for C_2 is found from Figure 5-6(a), and the distributive law is applied to obtain

$$\begin{aligned} C_2 &= G_1 + P_1(G_0 + P_0C_0) \\ &= G_1 + P_1G_0 + P_1P_0C_0 \end{aligned}$$

This equation is implemented by the logic with output C_2 in Figure 5-6(b). We obtain the two-level logic for C_3 by finding its equation from the carry path in Figure 5-6(a) and applying the distributive law:

$$C_3 = G_2 + P_2(G_1 + P_1(G_0 + P_0C_0))$$



□ **FIGURE 5-6**
Development of a Carry Lookahead Adder

$$\begin{aligned}
 &= G_2 + P_2(G_1 + P_1G_0 + P_0C_0) \\
 &= G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0
 \end{aligned}$$

The two-level logic with output C_3 in Figure 5-6(b) implements this function.

We could implement C_4 using the same method. But some of the gates would have a fan-in of five, which may increase the delay. Also, we are interested in reusing this same circuit for higher numbered bits (e.g., 4 through 7, 8 through 11, and 12 through 15 of a 16-bit adder). For this adder, in positions 4, 8, and 12 we would like the carry to be produced as fast as possible without using excessive fan-in. Accordingly, we want to repeat the same carry lookahead trick for *4-bit groups* that we used to handle the 4 bits. This will allow us to reuse the carry lookahead circuit for each group of 4 bits, and also to use the same circuit for four 4-bit groups as if they were individual bits. So instead of generating C_4 , we produce generate and propagate functions that apply to 4-bit groups instead of a single bit to act as the inputs for the group carry lookahead circuit. To propagate a carry from C_0 to C_4 , we need to have all four of the propagate functions equal to 1, giving the *group propagate* function

$$P_{0-3} = P_3P_2P_1P_0$$

To represent the generation of a carry in positions 0, 1, 2, and 3, and its propagation to C_4 , we need to consider the generation of a carry in each of the positions, as represented by G_0 through G_3 , and the propagation of each of these four generated carries to position 4. This gives the *group generate* function

$$G_{0-3} = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$$

The group propagate and group generate equations are implemented by the logic in the lower part of Figure 5-6(b). If there are only 4 bits in the adder, then the logic circuit used for C_1 can be used to generate C_4 from these two outputs. In a longer adder, a carry lookahead circuit identical to that in the figure, except for labeling, is placed at the second level to generate C_4 , C_8 , and C_{12} . This concept can be extended with more carry lookahead circuits in the second level and with one carry lookahead circuit in the third level to generate carries for positions 16, 32, and 48 in a 64-bit adder.

Assuming that an exclusive OR contributes 2 gate delays, the longest delay in the 4-bit carry lookahead adder is 6 gate delays, compared with 10 gate delays in the ripple carry adder. The improvement is very modest and perhaps not worth all the extra logic. But applying the carry lookahead circuit to a 16-bit adder using five copies in two levels of lookahead reduces the delay from 34 to just 10 gate delays, improving the performance of the adder by a factor of close to three. In a 64-bit adder, with the use of 21 carry lookahead circuits in three levels of lookahead, the delay is reduced from 130 gate delays to 14 gate delays, giving more than a factor of 8 in improved performance. In general, for the implementation we have shown, the delay of a carry lookahead adder designed for the best performance is $4L + 2$ gate delays, where L is the number of lookahead levels in the design.

5-3 BINARY SUBTRACTION

In Chapter 1, we briefly examined the subtraction of unsigned binary numbers. Although beginning texts cover only signed number addition and subtraction, to the complete exclusion of the unsigned alternative, unsigned number arithmetic plays an important role in computation and computer hardware design. It is used in floating-point units, in signed-magnitude addition and subtraction algorithms, and in extending the precision of fixed-point numbers. For these reasons, we will treat unsigned number addition and subtraction here. We also, however, choose to treat it first so that we can clearly justify, in terms of hardware cost, that which otherwise appears bizarre and often is accepted on faith, namely, the use of complement representations in arithmetic.

In Section 1-3, subtraction is performed by comparing the subtrahend with the minuend and subtracting the smaller from the larger. The use of a method containing this comparison operation results in inefficient and costly circuitry. As an alternative, we can simply subtract the subtrahend from the minuend. Using the same numbers as in a subtraction example from Section 1-3, we have

Borrows into:	11100
Minuend:	10011
Subtrahend:	<u>-11110</u>
Difference:	10101
Correct Difference:	-01011

If no borrow occurs into the most significant position, then we know that the subtrahend is not larger than the minuend and that the result is positive and correct. If a borrow does occur into the most significant position, as indicated in blue, then we know that the subtrahend is larger than the minuend. The result must then be negative, and so we need to correct its magnitude. We can do this by examining the result of the calculation when a borrow occurs:

$$M - N + 2^n$$

Note that the added 2^n represents the value of the borrow into the most significant position. Instead of this result, the desired magnitude is $N - M$. This can be obtained by subtracting the preceding formula from 2^n :

$$2^n - (M - N + 2^n) = N - M$$

In the previous example, $100000 - 10101 = 01011$, which is the correct magnitude.

In general, the subtraction of two n -digit numbers, $M - N$, in base 2 can be done as follows:

1. Subtract the subtrahend N from the minuend M .
2. If no end borrow occurs, then $M \geq N$, and the result is nonnegative and correct.

3. If an end borrow occurs, then $N > M$, and the difference, $M - N + 2^n$, is subtracted from 2^n , and a minus sign is appended to the result.

Subtraction of a binary number from 2^n to obtain an n -digit result is called taking the *2's complement* of the number. So in step 3, we are taking the 2's complement of the difference $M - N + 2^n$. Use of the 2's complement in subtraction is illustrated by the following example.

EXAMPLE 5-1 Unsigned Binary Subtraction by 2's Complement Subtract

Perform the binary subtraction $01100100 - 10010110$. We have

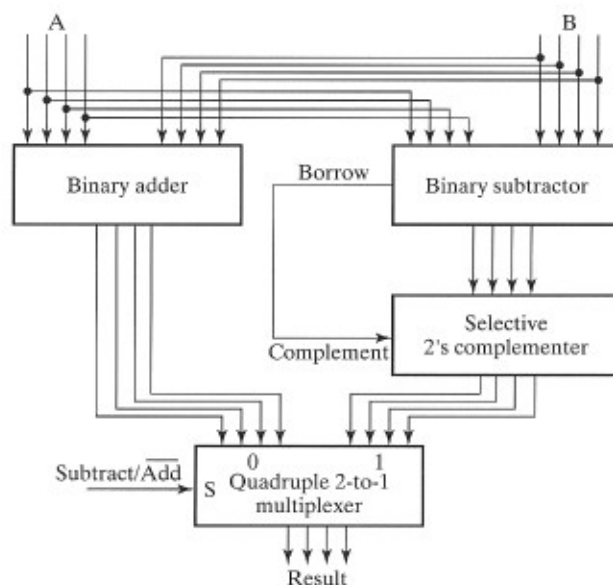
Borrows into:	10011110
Minuend:	01100100
Subtrahend:	<u>− 10010110</u>
Initial Result	11001110

The end borrow of 1 implies correction:

2^8	100000000
− Initial Result	<u>− 11001110</u>
Final Result	− 00110010 ■

To perform subtraction using this method requires a subtractor for the initial subtraction. In addition, when necessary, either the subtractor must be used a second time to perform the correction, or a separate 2's complementer circuit must be provided. So, thus far, we require a subtractor, an adder, and possibly a 2's complementer to perform both addition and subtraction. The block diagram for a 4-bit adder-subtractor using these functional blocks is shown in Figure 5-7. The inputs are applied to both the adder and the subtractor, so both operations are performed in parallel. If an end borrow value of 1 occurs in the subtraction, then the selective 2's complementer receives a value of 1 on its Complement input. This circuit then takes the 2's complement of the output of the subtractor. If the end borrow has value of 0, the selective 2's complementer passes the output of the subtractor through unchanged. If subtraction is the operation, then a 1 is applied to S of the multiplexer that selects the output of the complementer. If addition is the operation, then a 0 is applied to S , thereby selecting the output of the adder.

As we will see, this circuit is more complex than necessary. To reduce the amount of hardware, we would like to share logic between the adder and the subtractor. This can also be done using the notion of the complement. So before considering the combined adder-subtractor further, we will take a more careful look at complements.



□ **FIGURE 5-7**
Block Diagram of Binary Adder-Subtractor

Complements

There are two types of complements for each base- r system: the *radix complement*, which we saw earlier for base 2, and the *diminished radix complement*. The first is referred to as the r 's complement and the second as the $(r - 1)$'s complement. When the value of the base r is substituted in the names, the two types are referred to as the 2's and 1's complements for binary numbers and the 10's and 9's complements for decimal numbers, respectively. Since our interest for the present is in binary numbers and operations, we will deal with only 1's and 2's complements.

Given a number N in binary having n digits, the *1's complement* of N is defined as $(2^n - 1) - N$. 2^n is represented by a binary number that consists of a 1 followed by n 0's. $2^n - 1$ is a binary number represented by n 1's. For example, if $n = 4$, we have $2^4 = (10000)_2$ and $2^4 - 1 = (1111)_2$. Thus, the 1's complement of a binary number is obtained by subtracting each digit from 1. When subtracting binary digits from 1, we can have either $1 - 0 = 1$ or $1 - 1 = 0$, which causes the original bit to change from 0 to 1 or from 1 to 0, respectively. Therefore, the 1's complement of a binary number is formed by changing all 1's to 0's and all 0's to 1's—that is, applying the NOT or complement operation to each of the bits. Following are two numerical examples:

The 1's complement of 1011001 is 0100110.

The 1's complement of 0001111 is 1110000.

In similar fashion, the 9's complement of a decimal number, the 7's complement of an octal number, and the 15's complement of a hexadecimal number are obtained by subtracting each digit from 9, 7, and F (decimal 15), respectively.

Given an n -digit number N in binary, the *2's complement of N* is defined as $2^n - N$ for $N \neq 0$ and 0 for $N = 0$. The reason for the special case of $N = 0$ is that the result must have n bits, and subtraction of 0 from 2^n gives an $(n + 1)$ -bit result, $100\dots 0$. This special case is achieved by using only an n -bit subtractor or otherwise dropping the 1 in the extra position. Comparing with the 1's complement, we note that the 2's complement can be obtained by adding 1 to the 1's complement, since $2^n - N = [(2^n - 1) - N] + 1$. For example, the 2's complement of binary 101100 is $010011 + 1 = 010100$ and is obtained by adding 1 to the 1's complement value. Again, for $N = 0$, the result of this addition is 0, achieved by ignoring the carry out of the most significant position of the addition. These concepts hold for other bases as well. As we will see later, they are very useful in simplifying 2's complement and subtraction hardware.

Also, the 2's complement can be formed by leaving all least significant 0's and the first 1 unchanged and then replacing 1's with 0's and 0's with 1's in all other higher significant bits. Thus, the 2's complement of 1101100 is 0010100 and is obtained by leaving the two low-order 0's and the first 1 unchanged and then replacing 1's with 0's and 0's with 1's in the other four most significant bits. In other bases, the first nonzero digit is subtracted from the base r , and the remaining digits to the left are replaced with $r - 1$ minus their values.

It is also worth mentioning that the complement of the complement restores the number to its original value. To see this, note that the 2's complement of N is $2^n - N$, and the complement of the complement is $2^n - (2^n - N) = N$, giving back the original number.

Subtraction with Complements

Earlier, we expressed a desire to simplify hardware by sharing adder and subtractor logic. Armed with complements, we are prepared to define a binary subtraction procedure that uses addition and the corresponding complement logic. The subtraction of two n -digit unsigned numbers, $M - N$, in binary can be done as follows:

1. Add the 2's complement of the subtrahend N to the minuend M . This performs $M + (2^n - N) = M - N + 2^n$.
2. If $M \geq N$, the sum produces an end carry, 2^n . Discard the end carry, leaving result $M - N$.
3. If $M < N$, the sum does not produce an end carry since it is equal to $2^n - (N - M)$, the 2's complement of $N - M$. Perform a correction, taking the 2's complement of the sum and placing a minus sign in front to obtain the result $-(N - M)$.

The examples that follow further illustrate the foregoing procedure. Note that, although we are dealing with unsigned numbers, there is no way to get an unsigned result for the case in step 3. When working with paper and pencil, we recognize, by the absence of the end carry, that the answer must be changed to a negative number. If the minus sign for the result is to be preserved, it must be stored separately from the corrected n -bit result.

EXAMPLE 5-2 Unsigned Binary Subtraction by 2's Complement Addition

Given the two binary numbers $X = 1010100$ and $Y = 1000011$, perform the subtraction $X - Y$ and $Y - X$ using 2's complement operations. We have

$$\begin{array}{rcl}
 X & = & 1010100 \\
 \text{2's complement of } Y & = & 0111101 \\
 \text{Sum} & = & 10010001 \\
 \text{Discard end carry } 2^7 = - & \underline{10000000} & \\
 \text{Answer: } X - Y & = & 0010001 \\
 Y & = & 1000011 \\
 \text{2's complement of } X & = & \underline{0101100} \\
 \text{Sum} & = & 1101111
 \end{array}$$

There is no end carry.

$$\text{Answer: } Y - X = -(2\text{'s complement of } 1101111) = -0010001. \quad \blacksquare$$

Subtraction of unsigned numbers also can be done by means of the 1's complement. Remember that the 1's complement is one less than the 2's complement. Because of this, the result of adding the minuend to the complement of the subtrahend produces a sum that is one less than the correct difference when an end carry occurs. Discarding the end carry and adding one to the sum is referred to as an *end-around carry*.

EXAMPLE 5-3 Unsigned Binary Subtraction by 1's Complement Addition

Repeat Example 5-2 using 1's complement operations. Here, we have

$$\begin{array}{rcl}
 X - Y & = & 1010100 - 1000011 \\
 X & = & 1010100 \\
 \text{1's complement of } Y & = & + \underline{0111100} \\
 \text{Sum} & = & 10010000 \\
 \text{End-around carry} & \xrightarrow{\quad} & + 1 \\
 \text{Answer: } X - Y & = & 0010001
 \end{array}$$

$$\begin{aligned}
 Y - X &= 1000011 - 1010100 \\
 Y &= 1000011 \\
 \text{1's complement of } X &= + \underline{0101011} \\
 \text{Sum} &= 1101110
 \end{aligned}$$

There is no end carry.

Answer: $Y - X = -(1\text{'s complement of } 1101110) = -0010001.$ ■

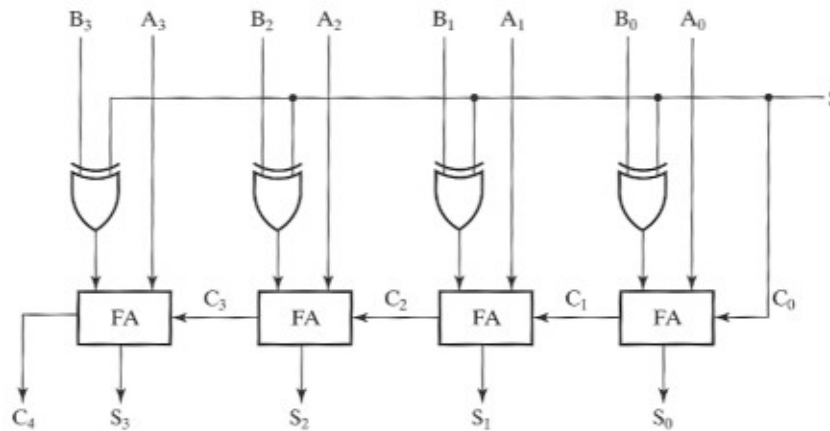
Note that the negative result is obtained by taking the 1's complement of the sum, since this is the type of complement being used.

5-4 BINARY ADDER-SUBTRACTORS

Using either the 2's or 1's complement, we have eliminated the subtraction operation and need only the appropriate complementer and an adder. When performing a subtraction we complement the subtrahend N , and when performing an addition we do not complement N . These operations can be accomplished by using a selective complementer and adder interconnected to form an adder-subtractor. We have used 2's complement, since it is most prevalent in modern systems. The 2's complement can be obtained by taking the 1's complement and adding 1 to the least significant bit. The 1's complement can be implemented easily with inverter circuits, and we can add 1 to the sum by making the input carry of the parallel adder equal to 1. Thus, by using 1's complement and an unused adder input, the 2's complement is obtained inexpensively. In 2's complement subtraction, as the correction step after adding, we complement the result and append a minus sign if an end carry does not occur. The correction operation is performed by using either the adder-subtractor a second time with $M = 0$ or a selective complementer as in Figure 5-7.

The circuit for subtracting $A - B$ consists of a parallel adder as shown in Figure 5-5, with inverters placed between each B terminal and the corresponding full-adder input. The input carry C_0 must be equal to 1. The operation that is performed becomes A plus the 1's complement of B plus 1. This is equal to A plus the 2's complement of B . For unsigned numbers, it gives $A - B$ if $A \geq B$ or the 2's complement of $B - A$ if $A < B$.

The addition and subtraction operations can be combined into one circuit with one common binary adder. This is done by including an exclusive-OR gate with each full adder. A 4-bit adder-subtractor circuit is shown in Figure 5-8. Input S controls the operation. When $S = 0$ the circuit is an adder, and when $S = 1$ the circuit becomes a subtractor. Each exclusive-OR gate receives input S and one of the inputs of B , B_i . When $S = 0$, we have $B_i \oplus 0$. If the full adders receive the value of B , and the input carry is 0, the circuit performs A plus B . When $S = 1$, we have $B_i \oplus 1 = \bar{B}_i$ and $C_0 = 1$. In this case, the circuit performs the operation A plus the 2's complement of B .



□ FIGURE 5-8
Adder-Subtractor Circuit

Signed Binary Numbers

In the previous section, we dealt with the addition and subtraction of unsigned numbers. We will now extend this approach to signed numbers, including a further use of complements that eliminates the correction step.

Positive integers and the number zero can be represented as unsigned numbers. To represent negative integers, we need a notation for negative values. In ordinary arithmetic, a negative number is indicated by a minus sign and a positive number by a plus sign. Because of hardware limitations, computers must represent everything with 1's and 0's, including the sign of a number. As a consequence, it is customary to represent the sign with a bit placed in the most significant position of an n -bit number. The convention is to make the sign bit 0 for positive numbers and 1 for negative numbers.

It is important to realize that both signed and unsigned binary numbers consist of a string of bits when represented in a computer. The user determines whether the number is signed or unsigned. If the binary number is signed, then the leftmost bit represents the sign and the rest of the bits represent the number. If the binary number is assumed to be unsigned, then the leftmost bit is the most significant bit of the number. For example, the string of bits 01001 can be considered as 9 (unsigned binary) or +9 (signed binary), because the leftmost bit is 0. Similarly, the string of bits 11001 represents the binary equivalent of 25 when considered as an unsigned number or -9 when considered as a signed number. The latter is because the 1 in the leftmost position designates a minus sign and the remaining four bits represent binary 9. Usually, there is no confusion in identifying the bits because the type of number representation is known in advance. The representation of signed numbers just discussed is referred to as the *signed-magnitude* system. In this system, the number consists of a magnitude and a symbol (+ or -) or a bit (0 or 1) indicating the sign. This is the representation of signed numbers used in ordinary arithmetic.

In implementing signed-magnitude addition and subtraction for n -bit numbers, the single sign bit in the leftmost position and the $n - 1$ magnitude bits are processed separately. The magnitude bits are processed as unsigned binary numbers. Thus, subtraction involves the correction step. To avoid this step, we use a different system for representing negative numbers, referred to as a *signed-complement* system. In this system, a negative number is represented by its complement. While the signed-magnitude system negates a number by changing its sign, the signed-complement system negates a number by taking its complement. Since positive numbers always start with 0 (representing a plus sign) in the leftmost position, their complements will always start with a 1, indicating a negative number. The signed-complement system can use either the 1's or the 2's complement, but the latter is the most common. As an example, consider the number 9, represented in binary with eight bits. +9 is represented with a sign bit of 0 in the leftmost position, followed by the binary equivalent of 9, to give 00001001. Note that all eight bits must have a value, and therefore, 0's are inserted between the sign bit and the first 1. Although there is only one way to represent +9, we have three different ways to represent -9 using eight bits:

In signed-magnitude representation:	10001001
In signed-1's complement representation:	11110110
In signed-2's complement representation:	11110111

In signed magnitude, -9 is obtained from +9 by changing the sign bit in the leftmost position from 0 to 1. In signed 1's complement, -9 is obtained by complementing all the bits of +9, including the sign bit. The signed 2's complement representation of -9 is obtained by taking the 2's complement of the positive number, including the 0 sign bit.

Table 5-3 lists all possible 4-bit signed binary numbers in the three representations. The equivalent decimal number is also shown. Note that the positive numbers in all three representations are identical and have 0 in the leftmost position. The signed 2's complement system has only one representation for 0, which is always positive. The other two systems have a positive 0 and a negative 0, which is something not encountered in ordinary arithmetic. Note that all negative numbers have a 1 in the leftmost bit position; this is the way we distinguish them from positive numbers. With 4 bits, we can represent 16 binary numbers. In the signed-magnitude and the 1's complement representations, there are seven positive numbers and seven negative numbers, and two signed zeros. In the 2's complement representation, there are seven positive numbers, one zero, and eight negative numbers.

The signed-magnitude system is used in ordinary arithmetic, but is awkward when employed in computer arithmetic due to the separate handling of the sign and the correction step required for subtraction. Therefore, the signed complement is normally used. The 1's complement imposes difficulties because of its two representations of 0 and is seldom used for arithmetic operations. It is useful as a logical operation, since the change from 1 to 0 or 0 to 1 is equivalent to a logical complement operation. The following discussion of signed binary arithmetic deals exclusively with the signed-2's complement representation of negative numbers because it prevails in actual use. By using 1's complementation and the end-around carry,

□ **TABLE 5-3**
Signed Binary Numbers

Decimal	Signed 2's Complement	Signed 1's Complement	Signed Magnitude
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000
-0	—	1111	1000
-1	1111	1110	1001
-2	1110	1101	1010
-3	1101	1100	1011
-4	1100	1011	1100
-5	1011	1010	1101
-6	1010	1001	1110
-7	1001	1000	1111
-8	1000	—	—

the same procedures as those for signed 2's complement can be applied to signed 1's complement.

Signed Binary Addition and Subtraction

The addition of two numbers, $M + N$, in the signed-magnitude system follows the rules of ordinary arithmetic: If the signs are the same, we add the two magnitudes and give the sum the sign of M . If the signs are different, we subtract the magnitude of N from the magnitude of M . The absence or presence of an end borrow then determines the sign of the result, based on the sign of M , and determines whether or not a 2's complement correction is performed. For example, since the signs are different, $(0\ 0011001) + (1\ 0100101)$ causes 0100101 to be subtracted from 0011001 . The result is 1110100 , and an end borrow of 1 occurs. The end borrow indicates that the magnitude of M is smaller than the magnitude of N . So the sign of the result is opposite that of M and is therefore a minus. The end borrow indicates that the magnitude of the result, 1110100 , must be corrected by taking its 2's complement. Combining the sign and the corrected magnitude of the result, we obtain $1\ 0001100$.

In contrast to this signed-magnitude case, the rule for adding numbers in the signed-complement system does not require comparison or subtraction, but only addition. The procedure is simple and can be stated as follows for binary numbers:

The addition of two signed binary numbers with negative numbers represented in signed-2's complement form is obtained from the addition of the two numbers, including their sign bits. A carry out of the sign bit position is discarded.

Numerical examples of signed binary addition are given in Example 5-4. Note that negative numbers will already be in 2's complement form and that the sum obtained after the addition, if negative, is left in that same form.

EXAMPLE 5-4 Signed Binary Addition Using 2's Complement

$$\begin{array}{rclclcl}
 +6 & 00000110 & -6 & 11111010 & +6 & 00000110 & -6 & 11111010 \\
 +13 & 00001101 & +13 & 00001101 & -13 & 11110011 & -13 & 11110011 \\
 +19 & 00010011 & +7 & 00000111 & -7 & 11111001 & -19 & 11101101
 \end{array}$$

In each of the four cases, the operation performed is addition, including the sign bits. Any carry out of the sign bit position is discarded, and negative results are automatically in 2's complement form. ■

The complement form for representing negative numbers is unfamiliar to people accustomed to the signed-magnitude system. To determine the value of a negative number in signed-2's complement, it is necessary to convert the number to a positive number in order to put it in a more familiar form. For example, the signed binary number 11111001 is negative because the leftmost bit is 1. Its 2's complement is 00000111, which is the binary equivalent of +7. We therefore recognize the original number to be equal to -7.

The subtraction of two signed binary numbers when negative numbers are in 2's complement form is very simple and can be stated as follows:

Take the 2's complement of the subtrahend (including the sign bit) and add it to the minuend (including the sign bit). A carry out of the sign bit position is discarded.

This procedure stems from the fact that a subtraction operation can be changed to an addition operation if the sign of the subtrahend is changed. That is,

$$(\pm A) - (+B) = (\pm A) + (-B)$$

$$(\pm A) - (-B) = (\pm A) + (+B)$$

But changing a positive number to a negative number is easily done by taking its 2's complement. The reverse is also true, because the complement of a negative number that is already in complement form produces the corresponding positive number. Numerical examples are shown in Example 5-5.

EXAMPLE 5-5 Signed Binary Subtraction Using 2's Complement

$$\begin{array}{rclclcl}
 -6 & 11111010 & 11111010 & +6 & 00000110 & 00000110 \\
 -(-13) & -11110011 & +00001101 & -(-13) & -11110011 & +00001101 \\
 +7 & & 00000111 & +19 & & 00010011
 \end{array}$$

The end carry is discarded. ■

It is worth noting that binary numbers in the signed-complement system are added and subtracted by the same basic addition and subtraction rules as are unsigned numbers. Therefore, computers need only one common hardware circuit to handle both types of arithmetic. The user or programmer must interpret the results of such addition or subtraction differently, depending on whether it is assumed that the numbers are signed or unsigned. Thus, the same adder-subtractor designed for unsigned numbers can be used for signed numbers.

If the signed numbers are in 2's complement representation, then the circuit in Figure 5-8 can be used with no correction step required. For 1's complement, the input from S to C_0 of the adder must be replaced by an input from C_n to C_0 .

Overflow

To obtain a correct answer when adding and subtracting, we must ensure that the result has a sufficient number of bits to accommodate the sum. If we start with two n -bit numbers, and the sum occupies $n + 1$ bits, we say that an *overflow* occurs. This is true for binary or decimal numbers, whether signed or unsigned. When one performs addition with paper and pencil, an overflow is not a problem, since we are not limited by the width of the page. We just add another 0 to a positive number and another 1 to a negative number, in the most significant position, to extend them to $n + 1$ bits and then perform the addition. Overflow is a problem in computers because the number of bits that hold a number is fixed, and a result that exceeds the number of bits cannot be accommodated. For this reason, computers detect and can signal the occurrence of an overflow. The overflow condition may be handled automatically by interrupting the execution of the program and taking special action. An alternative is to monitor for overflow conditions using software.

The detection of an overflow after the addition of two binary numbers depends on whether the numbers are considered to be signed or unsigned. When two unsigned numbers are added, an overflow is detected from the end carry out of the most significant position. In unsigned subtraction, the magnitude of the result is always equal to or smaller than the larger of the original numbers, making overflow impossible. In the case of signed 2's complement numbers, the most significant bit always represents the sign. When two signed numbers are added, the sign bit is treated as a part of the number, and an end carry of 1 does not necessarily indicate an overflow.

With signed numbers, an overflow cannot occur for an addition if one number is positive and the other is negative: Adding a positive number to a negative number produces a result whose magnitude is equal to or smaller than the larger of the original numbers. An overflow may occur if the two numbers added are both positive or both negative. To see how this can happen, consider the following 2's complement example: Two signed numbers, +70 and +80, are stored in two 8-bit registers. The range of binary numbers, expressed in decimal, that each register can accommodate is from +127 to -128. Since the sum of the two stored numbers is +150, it exceeds the capacity of an 8-bit register. This is also true for -70 and -80. These two additions, together with the two most significant carry bit values, are as follows:

Carries: 0 1		Carries: 1 0	
+ 70	0 1000110	- 70	1 0111010
+ 80	0 1010000	- 80	1 0110000
+ 150	1 0010110	- 150	0 1101010

Note that the 8-bit result that should have been positive has a negative sign bit and that the 8-bit result that should have been negative has a positive sign bit. If, however, the carry out of the sign bit position is taken as the sign bit of the result, then the 9-bit answer so obtained will be correct. But since there is no position in the result for the 9th bit, we say that an overflow has occurred.

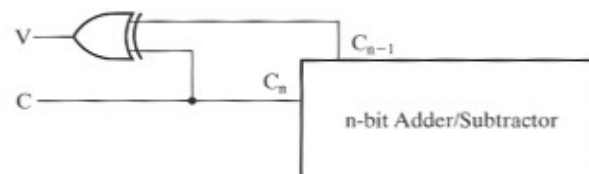
An overflow condition can be detected by observing the carry into the sign bit position and the carry out of the sign bit position. If these two carries are not equal, an overflow has occurred. This is indicated in the 2's complement example just completed, where the two carries are explicitly shown. If the two carries are applied to an exclusive-OR gate, an overflow is detected when the output of the gate is equal to 1. For this method to work correctly for 2's complement, it is necessary either to apply the 1's complement of the subtrahend to the adder and add 1 or to have overflow detection on the circuit that forms the 2's complement. The latter condition is due to overflow when complementing the maximum negative number.

Simple logic that provides overflow detection is shown in Figure 5-9. If the numbers are considered unsigned, then the C output being equal to 1 detects a carry (an overflow) for an addition and indicates that no correction step is required for a subtraction. C being equal to 0 detects no carry (no overflow) for an addition and indicates that a correction step is required for a subtraction.

If the numbers are considered signed, then the output V is used to detect an overflow. If $V = 0$ after a signed addition or subtraction, it indicates that no overflow has occurred and the result is correct. If $V = 1$, then the result of the operation contains $n + 1$ bits, but only the rightmost n of those bits fit in the n -bit result, so an overflow has occurred. The $(n+1)$ th bit is the actual sign, but it cannot occupy the sign bit position in the result.

5-5 BINARY MULTIPLICATION

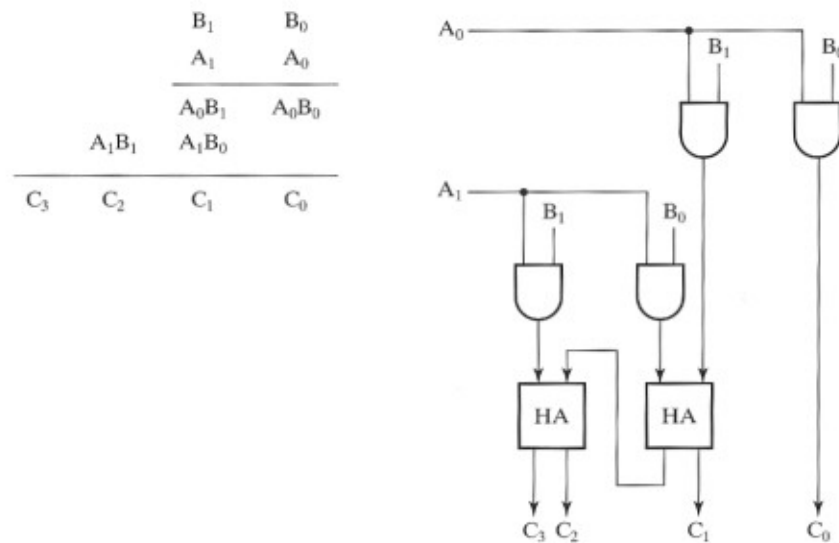
Multiplication of binary numbers is performed in the same way as with decimal numbers. The multiplicand is multiplied by each bit of the multiplier, starting from the least significant bit. Each such multiplication forms a partial product. Successive partial products are shifted one bit to the left. The final product is obtained from the sum of the partial products.



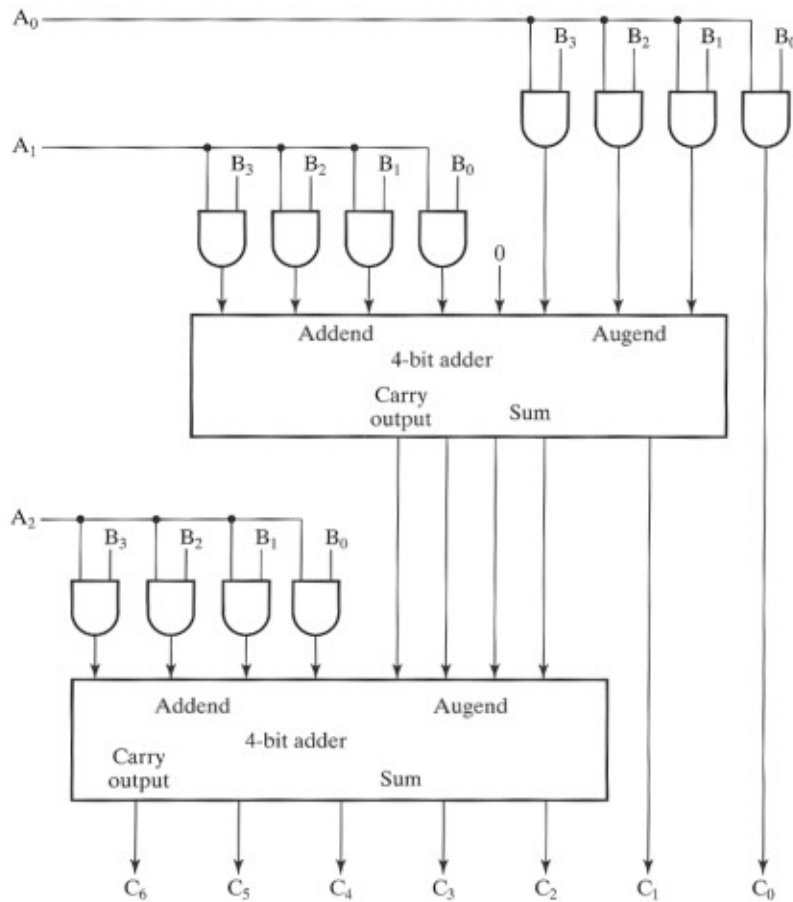
□ **FIGURE 5-9**
Overflow Detection Logic for Addition and Subtraction

To see how a binary multiplier can be implemented with a combinational circuit, consider the multiplication of two 2-bit numbers, as shown in Figure 5-10. The multiplicand bits are B_1 and B_0 , the multiplier bits are A_1 and A_0 , and the product is $C_3C_2C_1C_0$. The first partial product is formed by multiplying B_1B_0 by A_0 . The multiplication of two bits such as A_0 and B_0 produces a 1 if both bits are 1; otherwise it produces a 0. This is identical to an AND operation. Therefore, the partial product can be implemented with AND gates as shown in the diagram. The second partial product is formed by multiplying B_1B_0 by A_1 and is shifted one position to the left. The two partial products are added with two half-adder (HA) circuits. Usually there are more bits in the partial products, and it will be necessary to use full adders to produce the sum of the partial products. Note that the least significant bit of the product does not have to go through an adder, since it is formed by the output of the first AND gate.

A combinational circuit binary multiplier with more bits can be constructed in a similar fashion. A bit of the multiplier is ANDed with each bit of the multiplicand in as many levels as there are bits in the multiplier. The binary output in each level of AND gates is added in parallel with the partial product of the previous level to form a new partial product. The last level produces the product. For J multiplier bits and K multiplicand bits, we need $J \times K$ AND gates and $(J - 1) K$ -bit adders to produce a product of $J + K$ bits. As an example of a combinational circuit binary multiplier, consider a circuit that multiplies a binary number of four bits by a number of three bits. Let the multiplicand be represented by $B_3B_2B_1B_0$ and the multiplier by $A_2A_1A_0$. Since $K = 4$ and $J = 3$, we need 12 AND gates and two 4-bit adders to produce a product of 7 bits. The logic diagram of this kind of



□ FIGURE 5-10
A 2-Bit by 2-Bit Binary Multiplier



□ **FIGURE 5-11**
A 4-Bit by 3-Bit Binary Multiplier

multiplier circuit is shown in Figure 5-11. Note that the Carry Output bit enters the adder at the next level down in the multiplier.

5-6 OTHER ARITHMETIC FUNCTIONS

There are other arithmetic functions beyond $+$, $-$, and \times , that are quite important. Among these functions are incrementing, decrementing, multiplication and division by a constant, greater than comparison, and less than comparison. Each of these functions can be implemented for multiple-bit operands by using an iterative array of 1-bit cells. Instead of using these basic approaches, a combination of rudimentary functions and a new technique called contraction is used. Contraction begins with a circuit such as a binary adder, a carry-lookahead adder, or a binary multiplier. This approach simplifies design by converting existing circuits into useful, less-complicated circuits instead of designing the latter circuits directly.

Contraction

Value-fixing, transferring, and inverting on inputs can be combined with function blocks as done in Chapter 4 to implement new functions. We can implement new functions by using similar techniques on a given circuit or on its equations and then contracting it for a specific application to a simpler circuit. We will call the procedure *contraction*. The goal of contraction is to accomplish the design of a logic circuit or functional block by using results from past designs. It can be applied by the designer in designing a target circuit or can be applied by logic synthesis tools to simplify an initial circuit with value-fixing, transferring, and inverting on its inputs in order to obtain a target circuit. In both cases, contraction can also be applied to circuit outputs that are unused, to simplify a source circuit to a target circuit. First, we illustrate contraction by using Boolean equations.

EXAMPLE 5-6 Contraction of Full Adder Equations

The circuit Add1 to be designed is to form the sum S_i and carry C_{i+1} for the single bit addition $A_i + 1 + C_i$. This addition is a special case with $B_i = 1$ of the addition performed by a full adder, $A_i + B_i + C_i$. Thus, equations for the new circuit can be obtained by taking the full adder equations,

$$S_i = A_i \oplus B_i \oplus C_i$$

$$C_{i+1} = A_i B_i + A_i C_i + B_i C_i$$

setting $B_i = 1$, and simplifying the results, to obtain

$$S_i = A_i \oplus 1 \oplus C_i = \overline{A_i \oplus C_i}$$

$$C_{i+1} = A_i \cdot 1 + A_i C_i + 1 \cdot C_i = A_i + C_i$$

Suppose that this Add1 circuit is used in place of each of the four full adders in a 4-bit ripple carry adder. Instead of $S = A + B + C_0$, the computation being performed is $S = A + 1111 + C_0$. In 2's complement, this computation is $S = A - 1 + C_0$. If $C_0 = 0$, this implements the *decrement* operation $S = A - 1$, using considerably less logic than that used for a 4-bit addition or subtraction. ■

Contraction can be applied to equations, as done here, or directly on circuit diagrams with rudimentary functions applied to function block inputs. In order to successfully apply contraction, the desired function must be able to be obtained from the initial circuit by application of rudimentary functions on its inputs. Next we consider contraction based on unused outputs.

Placing an unknown value, X, on the output of a circuit means that output will not be used. Thus, the output gate and any other gates that drive only that output gate can be removed. The rules for contracting equations with X's on one or more outputs are as follows:

1. Delete all equations with X's on the circuit outputs.
2. If an intermediate variable does not appear in any remaining equation, delete its equation.

3. If an input variable does not appear in any remaining equation, delete it.
4. Repeat 2 and 3 until no new deletions are possible.

The rules for contracting a logic diagram with X's on one or more outputs are as follows:

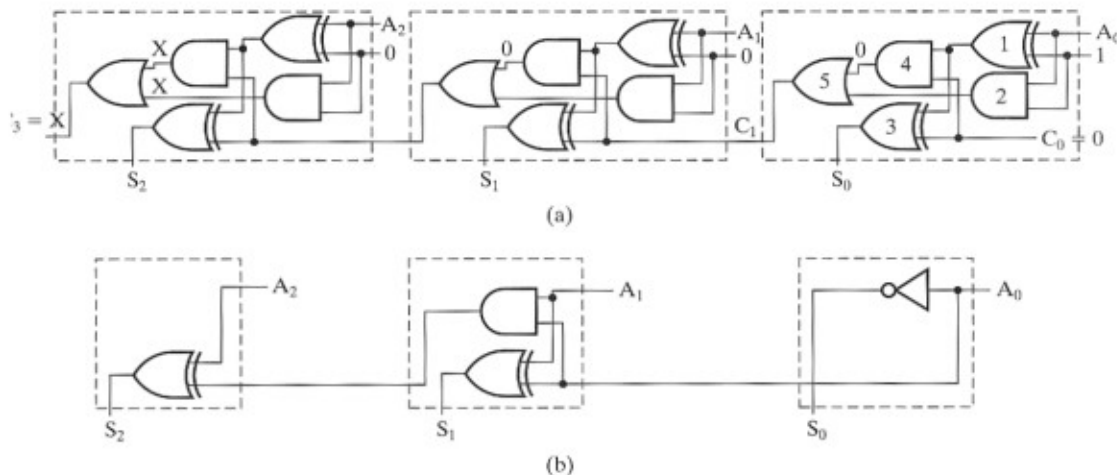
1. Beginning at the outputs, delete all gates with X's on their outputs and place X's on their input wires.
2. If all input wires driven by a gate are labeled with X's, delete the gate and place X's on its inputs.
3. If all input wires driven by an external input are labeled with X's, delete the input.
4. Repeat 2 and 3 until no new deletions are possible.

In the next subsection, contraction of a logic diagram is illustrated for the increment operation.

Incrementing

Incrementing means adding a fixed value to an arithmetic variable, most often a fixed value of 1. An n -bit *incrementer* that performs the operation $A + 1$ can be obtained by using a binary adder that performs the operation $A + B$ with $B = 0\dots01$. The use of $n = 3$ is large enough to determine the incrementer logic to construct the circuit needed for an n -bit incrementer.

Figure 5-12(a) shows a 3-bit adder with the inputs fixed to represent the computation $A + 1$ and with the output from the most significant carry bit C_3 fixed at value X. Operand $B = 001$ and the incoming carry $C_0 = 0$, so that $A + 001 + 0$ is computed. Alternatively, $B = 000$ and incoming carry $C_0 = 1$ could have been used.



□ **FIGURE 5-12**
Contraction of Adder to Incrementer

Based on value-fixing, there are three distinct contraction cases for the cells in the adder:

1. The least significant cell on the right with $B_0 = 1$ and $C_0 = 0$,
2. The typical cell in the middle with $B_1 = 0$, and
3. The most significant cell on the left with $B_2 = 0$ and $C_3 = X$.

For the right cell, the output of gate 1 becomes \bar{A}_0 so it can be replaced by an inverter. The output of gate 2 becomes A_0 , so it can be replaced by a wire connected to A_0 . Applying A_0 and 0 to gate 3, it can be replaced by a wire, connecting A_0 to the output S_0 . The output of gate 4 is 0, so it can be replaced with a 0 value. Applying this 0 and A_0 from gate 2 to gate 5, gate 5 can be replaced by a wire connecting A_0 to C_1 . The resulting circuit is shown as the right cell in Figure 5-12(b).

Applying the same technique to the typical cell with $B_1 = 0$ yields

$$S_1 = A_1 \oplus C_1$$

$$C_2 = A_1 C_1$$

giving the circuit shown as the middle cell in Figure 5-12(b). For the left cell with $B_2 = 0$ and $C_3 = X$, the effects of X are propagated first to save effort. Since gate A has X on its output, it is removed and X 's are placed on its two inputs. Since all gates driven by gates B and C have X 's on their inputs, they can be removed and X 's can be placed on their inputs. Gates D and E cannot be removed, since they are each driving a gate without an X on its input. The resulting circuit is shown as the left cell in Figure 5-12(b).

For an incrementer with $n > 3$ bits, the least significant incrementer cell is used in position 0, the typical cell in positions 1 through $n - 2$, and the most significant cell in position $n - 1$. In this example, the rightmost cell in position 1 is contracted, but, if desired, it could be replaced with the cell in position 2 with $B_0 = 0$ and $C_0 = 1$. Likewise, the output C_3 could be generated, but not used. In both cases, logic cost and power efficiency are sacrificed to make all of the cells identical.

Decrementing

Decrementing is the addition of a fixed negative value to an arithmetic variable, most often, a fixed value of -1 . A decrementer has already been designed in Example 5-6. Alternatively, a decrementer could be designed by using an adder-subtractor as a starting circuit and applying $B = 0...01$ and $C_0 = 0$, and selecting the subtraction operation by setting S to 1. Beginning with an adder-subtractor, we can also use contraction to design a circuit that increments for $S = 0$ and decrements for $S = 1$ by applying $B = 0...01$, $C_0 = 0$, and letting S remain a variable. In this case, the result is a cell of the complexity of a full adder in the typical bit positions. In fact, by going back to basics and redefining the carry function and designing the cell using this redefinition, the cost can be lowered somewhat. This illustrates that

contraction, while it yields an implementation, may not produce a result with the least cost or best performance.

Multiplication by Constants

Assuming the circuit in Figure 5-11 is used as a basis for multiplication, multiplication by a constant can be achieved by simply applying the constant as the multiplier A . If the value for a particular bit position is 1, then the multiplicand will be applied to an adder. If the value for a particular bit position is 0, then 0 will be applied to an adder and the adder will be removed by contraction. In both cases, the AND gates will be removed. The process is illustrated in Figure 5-13(a). For this case, the multiplier has been set to 101. In the contraction process, since $0 + B = B$, the carryout value is always 0. The end result of the contraction is a circuit that conveys the two least significant bits of B to the outputs C_1 and C_0 . The circuit adds the two most significant bits of B to B shifted two positions to the left and applies the result to product outputs C_6 through C_2 .

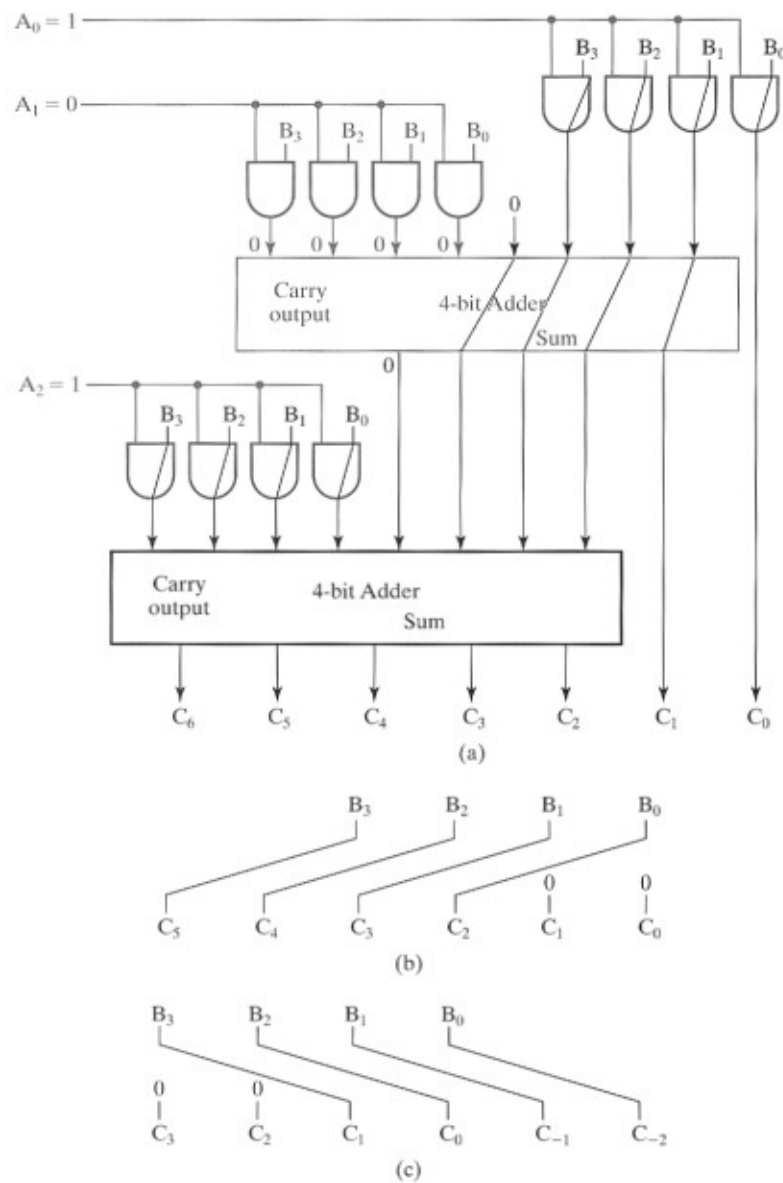
An important special case occurs when the constant equal 2^i (i.e., for multiplication $2^i \times B$). In this case, only one 1 appears in the multiplier and all logic is eliminated from the circuit resulting in only wires. In this case, for the 1 in position i , the result is B followed by i 0's. The functional block that results is simply a combination of skewed transfers and value fixing to 0. The function of this block is called a *left shift by i bit positions with zero fill*. *Zero fill* refers to the addition of 0's to the right of (or to the left of) an operand such as B . Shifting is a very important operation applied to both numerical and nonnumerical data. The contraction resulting from a multiplication by 2^2 (i.e., a left shift of 2 bit positions) is shown in Figure 5-13(b).

Division by Constants

Since we have not covered the division operation, our discussion of division by constants will be restricted to division by powers of 2 (i.e., by 2^i in binary). Since multiplication by 2^i results in addition of i 0's to the right of the multiplicand, by analogy, division by 2^i results in removal of the i least significant bits of the dividend. The remaining bits are the quotient and the bits discarded are the remainder. The function of this block is called a *right shift by i bit positions*. Just as for left shifting, right shifting is likewise a very important operation. The function block for division by 2^2 (i.e., right shifting by two bit positions) is shown in Figure 5-13 (c).

Zero Fill and Extension

Zero fill, as defined previously for multiplication by a constant, can also be used to increase the number of bits in an operand. For example, suppose that a byte 01101011 is to be used as an input to a circuit that requires an input of 16 bits. One possible way of producing the 16-bit input is to zero-fill with eight 0's on the left to produce 000000001101011. Another is to zero-fill on the right to produce



□ FIGURE 5-13

Contractions of Multiplier: (a) for $101 \times B$, (b) for $100 \times B$, and (c) for $B + 100$

0110101100000000. The former approach would be appropriate for operations such as addition or subtraction. The latter approach could be used to produce a low-precision 16-bit multiplication result in which the byte represents the most

significant eight bits of the actual product with the lower byte of the product discarded.

In contrast to zero fill, *sign extension* is used to increase the number of bits in an operand represented by using a complement representation for signed numbers. If the operand is positive, then bits can be added on the left by extending the sign of the number (0 for positive and 1 for negative). Byte 01101011, which represents 107 in decimal, extended to 16 bits becomes 000000001101011. Byte 10010101, which in 2's complement represents -107, extended to 16 bits becomes 111111110010101. The reason for using sign extension is to preserve the complement representation for signed numbers. For example, if 10010101 were extended with 0's, the magnitude represented would be very large, and further, the leftmost bit, which should be a 1 for a minus sign, would be incorrect in the 2's-complement representation.



DECIMAL ARITHMETIC The supplement that discusses decimal arithmetic functions and circuit implementations is available on the Companion Website for the text.

5-7 HDL REPRESENTATIONS—VHDL

Thus far, all of the VHDL descriptions used have contained only a single entity. Descriptions that represent circuits using hierarchies have multiple entities, one for each distinct element of the hierarchy, as shown in the next example.

EXAMPLE 5-7 Hierarchical VHDL for a 4-Bit Ripple Carry Adder

The example in Figures 5-14 and 5-15 uses three entities to build a hierarchical description of a 4-bit ripple carry adder. The style used for the architectures will be a mix of structural and dataflow description. The three entities are a half adder, a full adder that uses half adders, and the 4-bit adder itself. The architecture of `half_adder` consists of two dataflow assignments, one for `s` and one for `c`. The architecture of `full_adder` uses `half_adder` as a component. In addition, three internal signals, `hs`, `hc`, and `tc`, are declared. These signals are applied to two half adders and are also used in one dataflow assignment to construct the full adder in Figure 5-4. In the `adder_4` entity, four full adder components are simply connected together using the signals given in Figure 5-5.

Note that `C0` and `C4` are an input and an output, respectively, but `C(0)` through `C(4)` are internal signals (i.e., neither inputs nor outputs). `C(0)` is assigned `C0` and `C4` is assigned `C(4)`. The use of `C(0)` and `C(4)` separately from `C0` and `C4` is not essential here, but is useful to illustrate a VHDL constraint. Suppose we wanted to add overflow detection to the adder as shown in Figure 5-9. If `C(4)` is not defined separately, then one might attempt to write

```
V <= C(3) xor C4
```

```

-- 4-bit Adder: Hierarchical Dataflow/Structural
-- (See Figures 5-4 and 5-5 for logic diagrams)
library ieee;
use ieee.std_logic_1164.all;
entity half_adder is
    port (x, y : in std_logic;
          s, c : out std_logic);
end half_adder;

architecture dataflow_3 of half_adder is
begin
    s <= x xor y;
    c <= x and y;
end dataflow_3;

library ieee;
use ieee.std_logic_1164.all;
entity full_adder is
    port (x, y, z : in std_logic;
          s, c : out std_logic);
end full_adder;

architecture struc_dataflow_3 of full_adder is
    component half_adder
        port(x, y : in std_logic;
             s, c : out std_logic);
    end component;
    signal hs, hc, tc: std_logic;
begin
    HA1: half_adder
        port map (x, y, hs, hc);
    HA2: half_adder
        port map (hs, z, s, tc);
    c <= tc or hc;
end struc_dataflow_3;

library ieee;
use ieee.std_logic_1164.all;
entity adder_4 is
    port(B, A : in std_logic_vector(3 downto 0);
          C0 : in std_logic;
          S : out std_logic_vector(3 downto 0);
          C4: out std_logic);
end adder_4;

```

□ FIGURE 5-14

Hierarchical Structural/Dataflow Description of 4-Bit Full Adder

```

architecture structural_4 of adder_4 is
  component full_adder
    port(x, y, z : in std_logic;
         s, c : out std_logic);
  end component;
  signal C: std_logic_vector(4 downto 0);
begin
  Bit0: full_adder
    port map (B(0), A(0), C(0), S(0), C(1));
  Bit1: full_adder
    port map (B(1), A(1), C(1), S(1), C(2));
  Bit2: full_adder
    port map (B(2), A(2), C(2), S(2), C(3));
  Bit3: full_adder
    port map (B(3), A(3), C(3), S(3), C(4));
  C(0) <= C0;
  C4 <= C(4);
end structural_4;

```

□ FIGURE 5-15

Hierarchical Structural/Dataflow Description of 4-Bit Full Adder (Continued)

In VHDL, this is incorrect. An output cannot be used as an internal signal. Thus, it is necessary to define an internal signal to use in place of C4 (e.g., C(4)) giving

```
V <= C(3) xor C(4)
```

■

Behavioral Description

The 4-bit adder provides an opportunity to illustrate description of circuits at a levels higher than the logic level. Such levels of description are referred to as the behavioral level or the register transfer level. We will specifically study register transfers in Chapter 7. Without studying register transfers, however, we can still show a behavioral level description.

EXAMPLE 5-8 Behavioral VHDL for a 4-Bit Ripple Carry Adder

A behavioral description for the 4-bit adder is given in Figure 5-16. In the architecture of the entity `adder_4_b`, the addition logic is described by a single statement using + and &. The + represents addition and the & represents an operation called *concatenation*. A concatenation operator combines two signals into a single signal having its number of bits equal to the sum of the number of bits in the original signals. In the example, '0' & A represents the signal vector

```
'0' A(3) A(2) A(1) A(0)
```

```

-- 4-bit Adder: Behavioral Description
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity adder_4_b is
    port(B, A : in std_logic_vector(3 downto 0);
          C0 : in std_logic;
          S : out std_logic_vector(3 downto 0);
          C4: out std_logic);
end adder_4_b;

architecture behavioral of adder_4_b is
    signal sum : std_logic_vector(4 downto 0);
begin
    sum <= ('0' & A) + ('0' & B) + ("0000" & C0);
    C4 <= sum(4);
    S <= sum(3 downto 0);
end behavioral;

```

□ **FIGURE 5-16**
Behavioral Description of 4-Bit Full Adder

with $1 + 4 = 5$ signals. Note that '0', which appears on the left in the concatenation expression appears on the left in the signal listing. The inputs to the addition are all converted to 5-bit quantities for consistency, since the output including C4 is 5 bits. This conversion is not essential, but is a safe approach.

Since + cannot be performed on the `std_logic` type, we need an additional package to define addition for the `std_logic` type. In this case, we are using `std_logic_arith`, a package present in the `ieee` library. Further, we wish to specifically define the addition to be unsigned, so we use the `unsigned` extension. Also, concatenation in VHDL cannot be used on the left side of an assignment statement. To obtain C4 and S as the result of the addition, a 5-bit signal `sum` is declared. The signal `sum` is assigned the result of the addition including the carry out. Following are two additional assignment statements which split `sum` into outputs C4 and S. ■

This completes our introduction to VHDL for arithmetic circuits. We will continue with more on VHDL by presenting means for describing sequential circuits in Chapter 6.

5-8 HDL REPRESENTATIONS—VERILOG

Thus far, all of the descriptions used have contained only a single module. Descriptions that represent circuits using hierarchy have multiple modules, one for each distinct element of the hierarchy, as shown in the next example.

```
// 4-bit Adder: Hierarchical Dataflow/Structural
// (See Figures 5-4 and 5-5 for logic diagrams)
```

```
module half_adder_v(x, y, s, c);
    input x, y;
    output s, c;

    assign s = x ^ y;
    assign c = x & y;

endmodule

module full_adder_v(x, y, z, s, c);
    input x, y, z;
    output s, c;

    wire hs, hc, tc;

    half_adder_v    HA1(x, y, hs, hc),
                  HA2(hs, z, s, tc);
    assign c = tc | hc;

endmodule

module adder_4_v(B, A, C0, S, C4);
    input[3:0] B, A;
    input C0;
    output[3:0] S;
    output C4;

    wire[3:1] C;

    full_adder_v    Bit0(B[0], A[0], C0, S[0], C[1]),
                  Bit1(B[1], A[1], C[1], S[1], C[2]),
                  Bit2(B[2], A[2], C[2], S[2], C[3]),
                  Bit3(B[3], A[3], C[3], S[3], C4);

endmodule
```

□ **FIGURE 5-17**

Hierarchical Dataflow/Structural Verilog Description of 4-bit Adder

EXAMPLE 5-9 Hierarchical Verilog for a 4-Bit Ripple Carry Adder

The description in Figure 5-17 uses three modules to represent a hierarchical design for a 4-bit ripple carry adder. The style used for the modules will be a mix of structural and dataflow description. The three modules are a half adder, a full adder built around half adders, and the 4-bit adder itself.

The `half_adder` module consists of two dataflow assignments, one for `s` and one for `c`. The `full_adder` module uses the `half_adder` as a component as in Figure 5-4. In the `full_adder`, three internal wires, `hs`, `hc`, and `tc` are declared. Inputs, outputs and these wire names are applied to the two half adders

and `tc` and `hc` are ORed to form carry `c`. Note that the same names can be used on different modules (e.g., `x`, `y`, `s`, and `c` are used in both the `half_adder` and `full_adder`).

In the `adder_4` module, four full adders are simply connected together using the signals given in Figure 5-5. Note that `C0` and `C4` are an input and an output, respectively, but `C(3)` through `C(1)` are internal signals (i.e., neither inputs nor outputs). ■

Behavioral Description

The 4-bit adder provides an opportunity to illustrate description of circuits at a levels higher than the logic level. Such levels of description are referred to as the behavioral level or the register transfer level. We will specifically study register transfers in Chapter 7. Without studying register transfers, however, we can still show the behavioral level description for the 4-bit adder.

EXAMPLE 5-10 Behavioral Verilog for a 4-Bit Ripple Carry Adder

Figure 5-18 shows the Verilog description for the adder. In module `adder_4_b_v`, the addition logic is described by a single statement using `+` and `{}`. The `+` represent addition and the `{}` represent an operation called *concatenation*. The operation `+` performed on wire data types is unsigned. Concatenation combines two signals into a single signal having its number of bits equal to the sum of the number of bits in the original signals. In the example, `{C4, S}` represents the signal vector

$$C4\ S[3]\ S[2]\ S[1]\ S[0]$$

with $1 + 4 = 5$ signals. Note that `C4`, which appears on the left in the concatenation expression, appears on the left in the signal listing. ■

This completes our introduction to Verilog for arithmetic circuits. We will continue with more on Verilog by presenting means for describing sequential circuits in Chapter 6.

```
// 4-bit Adder: Behavioral Verilog Description

module adder_4_b_v(A, B, C0, S, C4);
    input [3:0] A, B;
    input C0;
    output [3:0] S;
    output C4;

    assign {C4, S} = A + B + C0;
endmodule
```

□ **FIGURE 5-18**
Behavioral Description of 4-Bit Full Adder Using Verilog

5-9 CHAPTER SUMMARY

This chapter introduced circuits for performing arithmetic. The implementation of binary adders, including the carry lookahead adder for improved performance, was treated in detail. The subtraction of unsigned binary numbers using 2's and 1's complements was presented, as was the representation of signed binary numbers and their addition and subtraction. The adder-subtractor, developed for unsigned binary, was found to apply directly to the addition and subtraction of signed 2's complement numbers as well. A very brief introduction to binary multiplication, using combinational circuits made up of AND gates and binary adders, was given.

Additional arithmetic operations introduced included incrementing, decrementing, multiplication and division by a constant, and shifting. The implementations for these operations were obtained by a design technique we called contraction. Zero fill and sign extension of operands was also introduced.

The last two sections of the chapter provided an introduction to VHDL and Verilog descriptions for arithmetic circuits. Both HDLs were illustrated by studying descriptions at the functional and behavioral levels for various functional blocks in the chapter.

REFERENCES

1. MANO, M. M. *Digital Design*, 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 2002.
2. WAKERLY, J. F. *Digital Design: Principles and Practices*, 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 2000.
3. *High-Speed CMOS Logic Data Book*. Dallas: Texas Instruments, 1989.
4. *IEEE Standard VHDL Language Reference Manual*. (ANSI/IEEE Std 1076-1993; revision of IEEE Std 1076-1987). New York: The Institute of Electrical and Electronics Engineers, 1994.
5. SMITH, D. J. *HDL Chip Design*. Madison, AL: Doone Publications, 1996.
6. PELLERIN, D. AND D. TAYLOR. *VHDL Made Easy!* Upper Saddle River, NJ: Prentice Hall PTR, 1997.
7. STEFAN, S. AND L. LINDH. *VHDL for Designers*. London: Prentice Hall Europe, 1997.
8. *IEEE Standard Description Language Based on the Verilog® Hardware Description Language* (IEEE Std 1364-1995). New York: The Institute of Electrical and Electronics Engineers, 1995.
9. PALNITKAR, S. *Verilog HDL: A Guide to Digital Design and Synthesis*. Upper Saddle River, NJ: SunSoft Press (A Prentice Hall Title), 1996.
10. BHASKER, J. *A Verilog HDL Primer*. Allentown, PA: Star Galaxy Press, 1997.
11. THOMAS, D., AND P. MOORBY. *The Verilog Hardware Description Language* 4th ed. Boston: Kluwer Academic Publishers, 1998.
12. CILETTI, M. *Modeling, Synthesis, and Rapid Prototyping with the Verilog HDL*, Upper Saddle River, NJ: Prentice Hall, 1999.

PROBLEMS



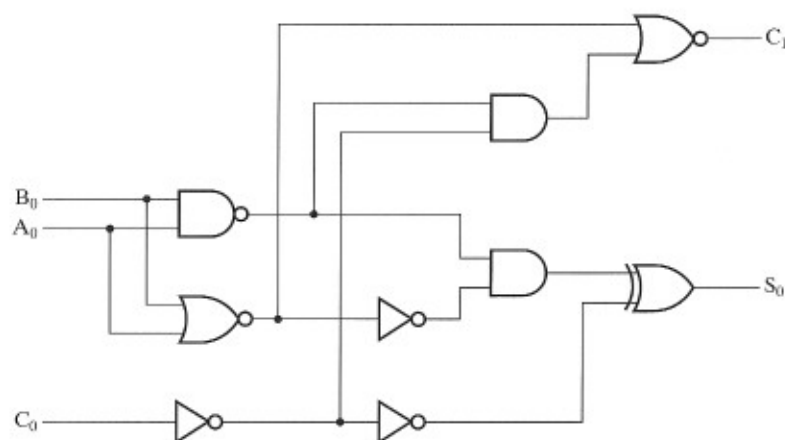
The plus (+) indicates a more advanced problem and the asterisk (*) indicates a solution is available on the Companion Website for the text.

- 5-1.** Design a combinational circuit that forms the 2-bit binary sum S_1S_0 of two 2-bit numbers A_1A_0 and B_1B_0 and has both a carry input C_0 and carry output C_2 . Design the entire circuit implementing each of the three outputs with a two-level circuit plus inverters for the input variables. Begin the design with the following equations for each of the two bits of the adder:

$$S_i = \overline{A_i}\overline{B_i}C_i + \overline{A_i}B_i\overline{C_i} + A_i\overline{B_i}\overline{C_i} + A_iB_iC_i$$

$$C_{i+1} = A_iB_i + A_iC_i + B_iC_i$$

- 5-2.** *The logic diagram of the first stage of a 4-bit adder, as implemented in integrated circuit type 74283, is shown in Figure 5-19. Verify that the circuit implements a full adder.
- 5-3.** *Obtain the 1's and 2's complements of the following unsigned binary numbers: 10011100, 10011101, 10101000, 00000000, and 10000000.
- 5-4.** Perform the indicated subtraction with the following unsigned binary numbers by taking the 2's complement of the subtrahend:
- (a) $11111 - 10000$ (c) $1011110 - 1011110$
 (b) $10110 - 1111$ (d) $101 - 101000$



□ **FIGURE 5-19**
Circuit for Problem 5-2

- 5-5.** Repeat problem 5-4, assuming the numbers are 2's complement signed numbers. Use extension to equalize the length of the operands. Does overflow occur during the complement operations for any of the given numbers? Does overflow occur for the overall subtraction for any of the given numbers?
- 5-6.** *Perform the arithmetic operations $(+36) + (-24)$ and $(-35) - (-24)$ in binary using signed-2's complement representation for negative numbers.
- 5-7.** The following binary numbers have a sign in the leftmost position and, if negative, are in 2's complement form. Perform the indicated arithmetic operations and verify the answers.
- (a) $100111 + 111001$ (c) $110001 - 010010$
 (b) $001011 + 100110$ (d) $101110 - 110111$

Indicate if overflow occurs for each computation.

- 5-8.** +Design three versions of the combinational circuit whose input is a 4-bit number and whose output is the 2's complement of the input number, for each of the following cases:
- (a) The circuit is a simplified two-level circuit, plus inverters as needed for the input variables.
- (b) The circuit is made up of four identical two-input, two-output cells, one for each bit. The cells are connected in cascade, with lines similar to a carry between them. The value applied to the rightmost carry bit is 0.
- (c) The circuit is redesigned with carry lookahead-like logic in order to speed up the circuit in part (b) for use in larger circuits with $4n$ input bits.
- 5-9.** Use contraction beginning with a 4-bit adder with carry out to design a 4-bit increment-by-2 circuit with carry out that adds the binary value 0010 to its 4-bit input. The function to be implemented is $S = A + 0010$.
- 5-10.** Use contraction beginning with an 8-bit adder-subtractor without carry out to design an 8-bit circuit without carry out that increments its input by 00000010 for input $S = 0$ and decrements its input by 00000010 for input $S = 1$. Perform the design by designing the distinct 1-bit cells needed and indicating the type of cell use in each of the eight bit positions.
- 5-11.** +(a) Use contraction beginning with a 4-bit carry lookahead adder with input carry and group carry and group propagate functions to design a 4-bit carry-lookahead-based circuit that increments its 4-bit input by the binary equivalent of 2.
- (b) Repeat part (a), designing a 4-bit carry-lookahead-based circuit that adds 0000 to its 4-bit input.
- (c) Construct a 16-bit carry-lookahead circuit that increments its 16-bit input by 2, giving the 16-bit output plus an output carry, by using the circuits designed in (a) and (b) an additional contraction of a 4-bit

carry-lookahead circuit that produces C_4 , C_8 , and C_{12} and G_{0-15} and P_{0-15} with inputs P_{0-3} , G_{0-3} , P_{4-7} , P_{8-11} , and P_{12-15} , and the additional logic needed to produce C_{16} . (Note that, due to part (b), G_{4-7} , G_{8-11} , and G_{12-15} are equal to 0.)

- 5-12.** Design a combinational circuit that compares two 4-bit unsigned numbers A and B to see whether B is greater than A . The circuit has one output X , so that $X = 1$ if $A < B$ and $X = 0$ if $A \geq B$.
- 5-13.** +Repeat Problem 5-12 by using three-input, one-output circuits, one for each of the four bits. The four circuits are connected together in cascade by carry-like signals. One of the inputs to each cell is a carry input, and the single output is a carry output.
- 5-14.** Repeat Problem 5-12 by applying contraction to a 4-bit subtractor and using the borrow out as X .
- 5-15.** Design a combinational circuit that compares 4-bit unsigned numbers A and B to see whether $A = B$ or $A < B$. Use an iterative circuit as in Problem 5-14.
- 5-16.** +Design a 5-bit signed-magnitude adder-subtractor. Divide the circuit for design into (1) sign generation and add-subtract control logic, (2) an unsigned number adder-subtractor using 2's complement of the minuend for subtraction, and (3) selective 2's complement result correction logic.
- 5-17.** *The adder-subtractor circuit of Figure 5-8 has the following values for input select S and data inputs A and B :

	S	A	B
(a)	0	0111	0111
(b)	1	0100	0111
(c)	1	1101	1010
(d)	0	0111	1010
(e)	1	0001	1000

Determine, in each case, the values of the outputs S_3 , S_2 , S_1 , S_0 , and C_4 .

- 5-18.** *Design a binary multiplier that multiplies two 4-bit unsigned numbers. Use AND gates and binary adders.
- 5-19.** Design a circuit that multiplies a 4-bit multiplicand by the constant 1010 by applying contraction to the solution to Problem 5-18.
- 5-20.** (a) Design a circuit that multiplies a 4-bit multiplicand by the constant 1000.
 (b) Design a circuit that divides an 8-bit dividend by the constant 1000 giving both an 8-bit quotient and an 8-bit remainder.



All files referred to in the remaining problems are available in ASCII form for simulation and editing on the Companion Website for the text. A VHDL or Verilog compiler/simulator is necessary for the problems or portions of problems requesting simulation. Descriptions can still be written, however, for many problems without using compilation or simulation.

- 5-21.** Compile and simulate the 4-bit adder in Figures 5-14 and 5-15. Apply combinations that check out the rightmost full adder for all eight input combinations; this also serves as a check for the other full adders. Also, apply combinations that check the carry chain connections between all full adders by demonstrating that a 0 and a 1 can be propagated from C_0 to C_4 .
- 5-22.** *Compile and simulate the behavioral description of the 4-bit Adder in Figure 5-16. Assuming a ripple carry implementation, apply combinations that check out the rightmost full adder for all eight input combinations. Also apply combinations that check the carry chain connections between all full adders by demonstrating that a 0 and a 1 can be propagated from C_0 to C_4 .
- 5-23.** +Using Figure 5-16 as a guide and a “when else” on S , write a high-level behavior VHDL description for the adder-subtractor in Figure 5-8. Compile and simulate your description. Assuming a ripple carry implementation, apply combinations that check out one of the full adder-subtractor stages for all 16 possible input combinations. Also, apply combinations to check the carry chain connections in between the full adders by demonstrating that a 0 and a 1 can be propagated from C_0 to C_4 .
- 5-24.** +Write a hierarchical dataflow VHDL description similar to that in Figures 5-14 and 5-15 for the logic of the 4-bit carry lookahead adder in Figure 5-6. Compile and simulate your description. Devise a set of tests that does a good job of exercising the logic in the adder.
- 5-25.** Compile and simulate the 4-bit Adder in Figure 5-17. Apply combinations that check out the rightmost full adder for all eight input combinations; this also serves as a check for the other full adders. Also, apply combinations that check the carry chain connections between all full adders by demonstrating that a 0 and a 1 can be propagated from C_0 to C_4 .
- 5-26.** *Compile and simulate the behavioral description of the 4-bit adder in Figure 5-18. Assuming a ripple carry implementation, apply all eight input combinations to check out the rightmost full adder. Also, apply combinations to check the carry chain connections between all full adders by demonstrating that a 0 and a 1 can be propagated from C_0 to C_4 .
- 5-27.** Using Figure 5-18 as a guide and a “binary decision” on S from Figure 4-37, write a high-level behavior Verilog description for the adder-subtractor in Figure 5-8. Compile and simulate your description. Assuming a ripple carry implementation, apply input combinations to your design that will (1) cause all 16 possible input combinations to be applied to the full adder-subtractor stage for bit 2, and (2) simultaneously cause the carry output of bit 2 to

appear at one of your design's outputs. Also, apply combinations that check the carry chain connections between all full adders by demonstrating that a 0 and a 1 can be propagated from C0 to C4.

- 5-28.** +Write a hierarchical Verilog dataflow description similar to that in Figure 5-17 for the logic of the 4-bit carry lookahead adder in Figure 5-6. Compile and simulate your description. Devise a set of tests that does a good job of exercising the logic in the adder.

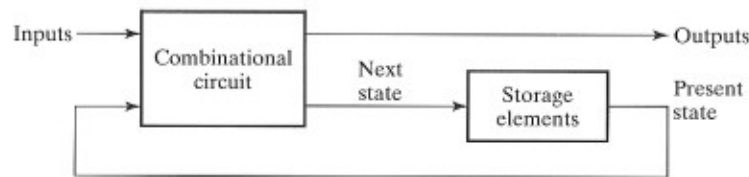
CHAPTER

6

SEQUENTIAL CIRCUITS

To this point, we have studied only combinational logic. Although such logic is capable of interesting operations, such as addition and subtraction, the performance of useful sequences of operations using combinational logic alone requires cascading many structures together. The hardware to do this, however, is very costly and inflexible. In order to perform useful or flexible sequences of operations, we need to be able to construct circuits that can store information between the operations. Such circuits are called sequential circuits. This chapter begins with an introduction to sequential circuits, which is followed by a study of the basic elements for storing binary information, called latches and flip-flops. We distinguish flip-flops from latches and study various types of each. We then analyze sequential circuits consisting of both flip-flops and combinational logic. State tables and state diagrams provide a means for describing the behavior of sequential circuits. Subsequent sections of the chapter develop the techniques for designing sequential circuits and verifying their correctness. In the last two sections, we provide VHDL and Verilog hardware description language representations for storage elements and for the type of sequential circuits in this chapter.

Latches, flip-flops, and sequential circuits are fundamental components in the design of almost all digital logic. In the generic computer given at the beginning of Chapter 1, latches and flip-flops are widespread in the design. The exception is memory circuits, since large portions of memory are designed as electronic circuits rather than as logic circuits. Nevertheless, due to the wide use of logic-based storage, this chapter contains fundamental material for any in-depth understanding of computers and digital systems and how they are designed.



□ **FIGURE 6-1**
Block Diagram of a Sequential Circuit

6-1 SEQUENTIAL CIRCUIT DEFINITIONS

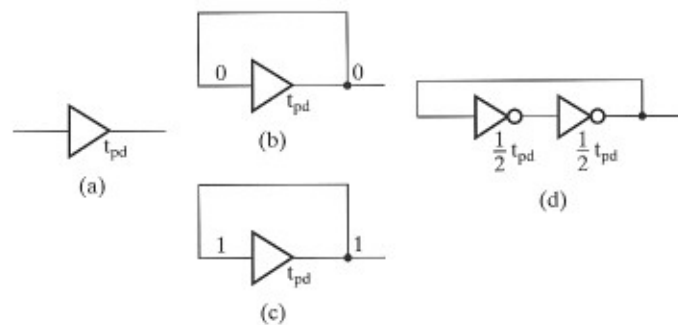
The digital circuits considered thus far have been combinational. Although every digital system is likely to include a combinational circuit, most systems encountered in practice also include storage elements, requiring that the systems be described as sequential circuits.

A block diagram of a sequential circuit is shown in Figure 6-1. A combinational circuit and storage elements are interconnected to form the sequential circuit. The storage elements are circuits that are capable of storing binary information. The binary information stored in these elements at any given time defines the *state* of the sequential circuit at that time. The sequential circuit receives binary information from its environment via the inputs. These inputs, together with the present state of the storage elements, determine the binary value of the outputs. They also determine the values used to specify the next state of the storage elements. The block diagram demonstrates that the outputs in a sequential circuit are a function not only of the inputs, but also of the present state of the storage elements. The next state of the storage elements is also a function of the inputs and the present state. Thus, a sequential circuit is specified by a time sequence of inputs, internal states, and outputs.

There are two main types of sequential circuits, and their classification depends on the times at which their inputs are observed and their internal state changes. The behavior of a *synchronous sequential circuit* can be defined from the knowledge of its signals at discrete instants of time. The behavior of an *asynchronous sequential circuit* depends upon the inputs at any instant of time and the order in continuous time in which the inputs change.

Information is stored in digital systems in many ways, including the use of logic circuits. Figure 6-2 (a) shows a buffer. This buffer has a propagation delay t_{pd} . Since information present at the buffer input at time t appears at the buffer output at time $t + t_{pd}$, the information has effectively been stored for time t_{pd} . But, in general, we wish to store information for an indefinite time that is typically much longer than the time delay of one or even many gates. This stored value is to be changed at arbitrary times based on the inputs applied to the circuit and should not depend on the specific time delay of a gate.

Suppose that the output of the buffer in Figure 6-2(a) is connected to its input as shown in Figures 6-2(b) and (c). Suppose further that the value on the input to the buffer in part (b) has been 0 for at least time t_{pd} . Then the output produced by the buffer will be 0 at time $t + t_{pd}$. This output is applied to the input so



□ **FIGURE 6-2**
Logic Structures for Storing Information

that the output will also be 0 at time $t + 2 t_{pd}$. This relationship between input and output holds for all t , so the 0 will be stored indefinitely. The same argument can be made for storing a 1 in the circuit in Figure 6-2(c).

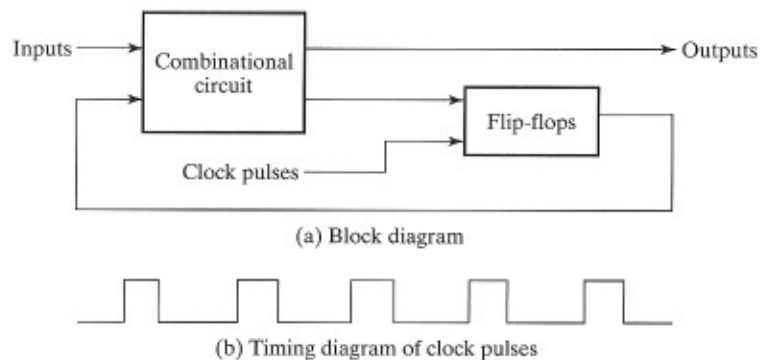
The example of the buffer illustrates that storage can be constructed from logic with delay connected in a closed loop. Any loop that produces such storage must also have a property possessed by the buffer, namely, that there must be no inversion of the signal around the loop. A buffer is usually implemented by using two inverters, as shown in Figure 6-2(d). The signal is inverted twice, that is,

$$\overline{\overline{X}} = X$$

giving no net inversion of the signal around the loop. In fact, this example is an illustration of one of the most popular methods of implementing storage in computer memories. (See Chapter 9.) However, although the circuits in Figures 6-2(b) through (d) are able to store information, there is no way for the information to be changed. By replacing the inverters with NOR or NAND gates, the information can be changed. Asynchronous storage circuits called latches are made in this manner and are discussed in the next section.

In general, more complex asynchronous circuits are difficult to design, since their behavior is highly dependent on the propagation delays of the gates and on the timing of the input changes. Thus, circuits that fit the synchronous model are the choice of most designers. Nevertheless, some asynchronous design is necessary. A very important case is the use of asynchronous latches as blocks to build storage elements, called flip-flops, that store information in synchronous circuits.

A synchronous sequential circuit employs signals that affect the storage elements only at discrete instants of time. Synchronization is achieved by a timing device called a *clock generator* which produces a periodic train of *clock pulses*. The pulses are distributed throughout the system in such a way that synchronous storage elements are affected only in some specified relationship to every pulse. In practice, the clock pulses are applied with other signals that specify the required change in the storage elements. The outputs of storage elements can change their value only in the presence of clock pulses. Synchronous sequential circuits that use clock pulses as inputs to storage elements are called *clocked sequential circuits*. These are the type of



□ **FIGURE 6-3**
Synchronous Clocked Sequential Circuit

circuit most frequently encountered in practice, since they operate correctly in spite of wide differences in circuit delays and are relatively easy to design.

The storage elements used in clocked sequential circuits are called flip-flops. For simplicity, assume circuits with a single clock signal. A *flip-flop* is a binary storage device capable of storing one bit of information and having timing characteristics to be defined in Section 6-3. The block diagram of a synchronous clocked sequential circuit is shown in Figure 6-3. The flip-flops receive their inputs from the combinational circuit and also from a clock signal with pulses that occur at fixed intervals of time, as shown in the timing diagram. The flip-flops can change state only in response to a clock pulse. For synchronous operation, when a clock pulse is absent, the flip-flop outputs cannot change even if the outputs of the combinational circuit driving their inputs change in value. Thus, the feedback loops shown in the figure between the combinational logic and the flip-flops are broken. As a result, a transition from one state to the other occurs only at fixed time intervals dictated by the clock pulses, giving synchronous operation. The sequential circuit outputs are shown as outputs of the combinational circuit. This is valid even when some sequential circuit outputs are actually the flip-flop outputs. In this case, the combinational circuit part between the flip-flop outputs and the sequential circuit outputs consists of connections only.

A flip-flop has one or two outputs, one for the normal value of the bit stored and an optional one for the complemented value of the bit stored. Binary information can enter a flip-flop in a variety of ways, a fact that gives rise to different types of flip-flops. Our focus will be on the most prevalent flip-flop used today, the D flip-flop. In section 6-6, other flip-flop types will be considered. In preparation for studying flip-flops and their operation, necessary groundwork is presented in the next section on latches from which the flip-flops are constructed.

6-2 LATCHES

A storage element can maintain a binary state indefinitely (as long as power is delivered to the circuit), until directed by an input signal to switch states. The major differences among the various types of latches and flip-flops are the number

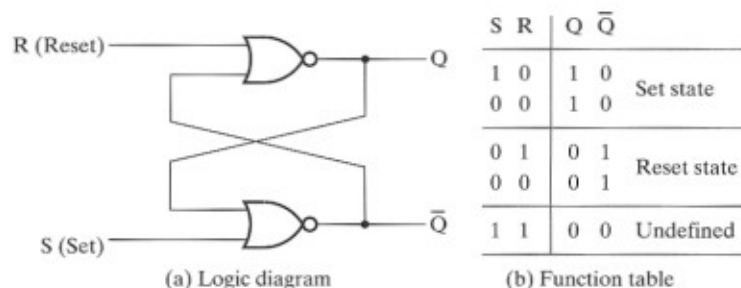
of inputs they possess and the manner in which the inputs affect the binary state. The most basic storage elements are latches, from which flip-flops are usually constructed. Although latches are most often used within flip-flops, they can also be used with more complex clocking methods to implement sequential circuits directly. The design of such circuits is, however, beyond the scope of the basic treatment given here. In this section, the focus is on latches as basic primitives for constructing storage elements.

SR and \overline{SR} Latches

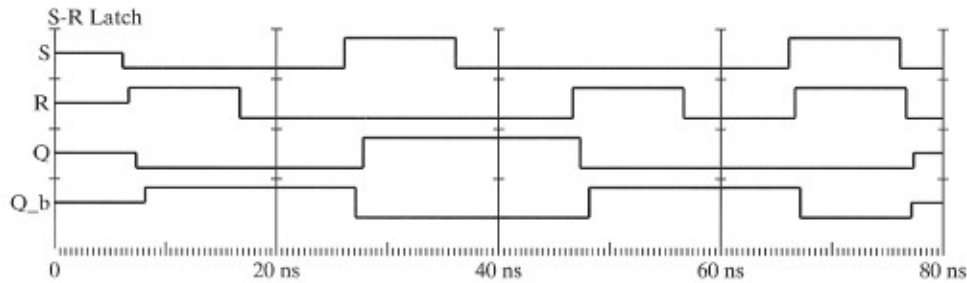
The SR latch is a circuit constructed from two cross-coupled NOR gates. It is derived from the single-loop storage element in Figure 6-2(d) by simply replacing the inverters with NOR gates, as shown in Figure 6-4(a). This replacement allows the stored value in the latch to be changed. The latch has two inputs, labeled S for set and R for reset, and two useful states. When output $Q = 1$ and $\overline{Q} = 0$, the latch is said to be in the *set state*. When $Q = 0$ and $\overline{Q} = 1$, it is in the *reset state*. Outputs Q and \overline{Q} are normally the complements of each other. When both inputs are equal to 1 at the same time, an undefined state with both outputs equal to 0 occurs.

Under normal conditions, both inputs of the latch remain at 0 unless the state is to be changed. The application of a 1 to the S input causes the latch to go to the set (1) state. The S input must go back to 0 before R is changed to 1 to avoid occurrence of the undefined state. As shown in the function table in Figure 6-4(b), two input conditions cause the circuit to be in the set state. The initial condition is $S = 1, R = 0$, to bring the circuit to the set state. Applying a 0 to S with $R = 0$ leaves the circuit in the same state. After both inputs return to 0, it is possible to enter the reset state by applying a 1 to the R input. The 1 can then be removed from R , and the circuit remains in the reset state. Thus, when both inputs are equal to 0, the latch can be in either the set or the reset state, depending on which input was most recently a 1.

If a 1 is applied to both the inputs of the latch, both outputs go to 0. This produces an undefined state because it violates the requirement that the outputs be the complement of each other. It also results in an indeterminate or unpredictable next state when both inputs return to 0 simultaneously. In normal operation, these problems are avoided by making sure that 1's are not applied to both inputs simultaneously.



□ **FIGURE 6-4**
 SR Latch with NOR Gates

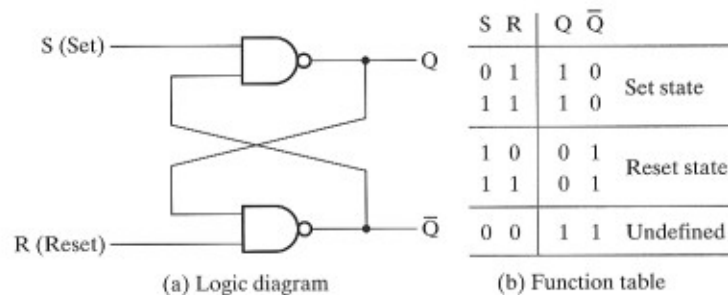


□ **FIGURE 6-5**
Logic Simulation of *SR* Latch Behavior

The behavior of the *SR* latch described in the preceding paragraph is illustrated by the ModelSim® logic simulator waveforms shown in Figure 6-5. Initially, the inputs and the state of the latch are unknown, as indicated by a logic level half-way between 0 and 1. When *R* becomes 1 with *S* at 0, the latch is reset with *Q* first becoming 0 and, in response, *Q_b* (which represents \overline{Q}) becomes 1. Next, when *R* becomes 0, the latch remains reset, storing the 0 value present on *Q*. When *S* becomes 1 with *R* at 0, the latch is set, with *Q_b* going to 0 first and, in response, *Q* going to 1 next. The delays in the changes of *Q* and *Q_b* after an input change are directly related to the delays of the two NOR gates used in the latch implementation. When *S* returns to 0, the latch remains set, storing the 1 value present on *Q*. When *R* becomes 1 with *S* equal to 0, the latch is reset with *Q* changing to 0 and *Q_b* responding by changing to 1. The latch remains reset when *R* returns to 0. When *S* and *R* both become 1, both *Q* and *Q_b* become 0. When *S* and *R* simultaneously return to 0, both *Q* and *Q_b* take on unknown values. This form of indeterminate state behavior for the ($\overline{S}, \overline{R}$) sequence of inputs (1, 1), (0, 0) results from assuming simultaneous input changes and equal gate delays. The actual indeterminate behavior that occurs depends on circuit delays and slight differences in the times at which *S* and *R* change in the actual circuit. Regardless of the simulation results, these indeterminate behaviors are viewed as undesirable, and the input combination (1,1) is avoided. In general, the latch state changes only in response to input changes and remains unchanged otherwise.

The $\overline{S}\overline{R}$ latch with two cross-coupled NAND gates is shown in Figure 6-6. It operates with both inputs normally at 1, unless the state of the latch has to be changed. The application of a 0 to the *S* input causes output *Q* to go to 1, putting the latch in the set state. When the *S* input goes back to 1, the circuit remains in the set state. With both inputs at 1, the state of the latch is changed by placing a 0 on the *R* input. This causes the circuit to go to the reset state and stay there even after both inputs return to 1. The condition that is undefined for this NAND latch is when both inputs are equal to 0 at the same time, an input combination that should be avoided.

Comparing the NAND latch with the NOR latch, note that the input signals for the NAND require the complement of those values used for the NOR. Because the NAND latch requires a 0 signal to change its state, it is referred to as an $\overline{S}\overline{R}$

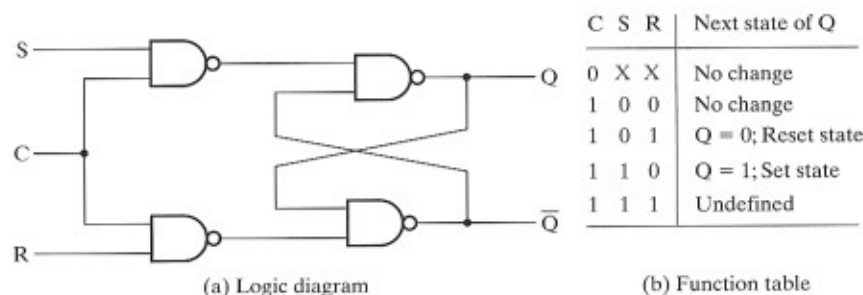


□ **FIGURE 6-6**
 $\bar{S}\bar{R}$ Latch with NAND Gates

latch. The bar above the letters designates the fact that the inputs must be in their complement form in order to act upon the circuit state.

The operation of the basic NOR and NAND latches can be modified by providing an additional control input that determines when the state of the latch can be changed. An SR latch with a control input is shown in Figure 6-7. It consists of the basic NAND latch and two additional NAND gates. The control input C acts as an enable signal for the other two inputs. The output of the NAND gates stays at the logic-1 level as long as the control input remains at 0. This is the quiescent condition for the $\bar{S}\bar{R}$ latch composed of two NAND gates. When the control input goes to 1, information from the S and R inputs is allowed to affect the $\bar{S}\bar{R}$ latch. The set state is reached with $S = 1$, $R = 0$, and $C = 1$. To change to the reset state, the inputs must be $S = 0$, $R = 1$, and $C = 1$. In either case, when C returns to 0, the circuit remains in its current state. Control input $C = 0$ disables the circuit so that the state of the output does not change, regardless of the values of S and R . Moreover, when $C = 1$ and both the S and R inputs are equal to 0, the state of the circuit does not change. These conditions are listed in the function table accompanying the diagram.

An undefined state occurs when all three inputs are equal to 1. This condition places 0's on both inputs of the basic $\bar{S}\bar{R}$ latch, giving an undefined state. When the control input goes back to 0, one cannot conclusively determine the next state, since



□ **FIGURE 6-7**
 SR Latch with Control Input

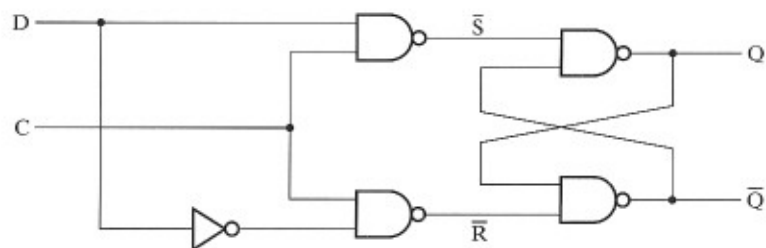
the *SR* latch sees inputs (0, 0) followed by (1, 1). The *SR* latch with control input is an important circuit because other latches and flip-flops are constructed from it. Sometimes the *SR* latch with control input is referred to as an *SR* (or *RS*) flip-flop; however, according to our terminology, it does not qualify as a flip-flop, since the circuit does not fulfill the flip-flop requirements presented in the next section.

D Latch

One way to eliminate the undesirable undefined state in the *SR* latch is to ensure that inputs *S* and *R* are never equal to 1 at the same time. This is done in the *D* latch, shown in Figure 6-8. This latch has only two inputs: *D* (data) and *C* (control). The complement of *D* input goes directly to the \bar{S} input, and *D* is applied to the \bar{R} input. As long as the control input is 0, the $\bar{S}\bar{R}$ latch has both inputs at the 1 level, and the circuit cannot change state regardless of the value of *D*. The *D* input is sampled when *C* = 1. If *D* is 1, the *Q* output goes to 1, placing the circuit in the set state. If *D* is 0, output *Q* goes to 0, placing the circuit in the reset state.

The *D* latch receives its designation from its ability to hold *data* in its internal storage. The binary information present at the data input of the *D* latch is transferred to the *Q* output when the control input is enabled (1). The output follows changes in the data input, as long as the control input is enabled. When the control input is disabled (0), the binary information that was present at the data input at the time the transition occurred is retained at the *Q* output until the control input is enabled again.

The *D* latch in VLSI circuits is often constructed with transmission gates (TGs), as shown in Figure 6-9. The TG was defined in Figure 2-35. The *C* input controls two TGs. When *C* = 1, the TG connected to input *D* conducts, and the TG connected to output *Q* disconnects. This produces a path from input *D* through two

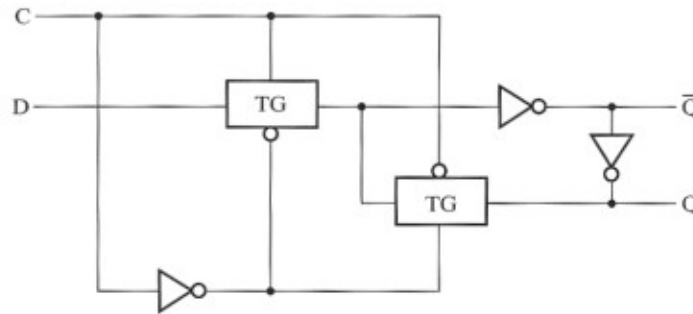


(a) Logic diagram

C	D	Next state of Q
0	X	No change
1	0	Q = 0; Reset state
1	1	Q = 1; Set state

(b) Function table

□ **FIGURE 6-8**
D Latch



□ **FIGURE 6-9**
D Latch with Transmission Gates

inverters to output Q . Thus, the output follows the data input as long as C remains active (1). When C changes to 0, the first TG disconnects input D from the circuit, and the second TG connects the two inverters at the output into a loop. Hence, the value that was present at input D at the time that C went from 1 to 0 is retained at the Q output by the loop.

6-3 FLIP-FLOPS

The state of a latch in a flip-flop is allowed to switch by a momentary change in value on the control input. This change is called a *trigger*, and it enables, or triggers, the flip-flop. The D latch with clock pulses on its control input is triggered every time a pulse to the logic-1 level occurs. As long as the pulse remains at the active (1) level, any changes in the data input will change the state of the latch. In this sense, the latch is *transparent*, since its input value can be seen from the outputs.

As the block diagram of Figure 6-3 shows, a sequential circuit has a feedback path from the outputs of the flip-flops to the combination circuit. As a consequence, the data inputs of the flip-flops are derived in part from the outputs of the same and other flip-flops. When latches are used for the storage elements, a serious difficulty arises. The state transitions of the latches start as soon as the clock pulse changes to the logic-1 level. The new state of a latch may appear at its output while the pulse is still active. This output is connected to the inputs of some of the latches through a combinational circuit. If the inputs applied to the latches change while the clock pulse is still in the logic-1 level, the latches will respond to *new state values* of other latches instead of the *original state values*, and a succession of changes of state instead of a single one may occur. The result is an unpredictable situation, since the state may keep changing and continue to change until the clock returns to 0. The final state depends on how long the clock pulse stays at level logic 1. Because of this unreliable operation, the output of a latch cannot be applied directly or through combinational logic to the input of the same or another latch when all the latches are triggered by a single clock signal.

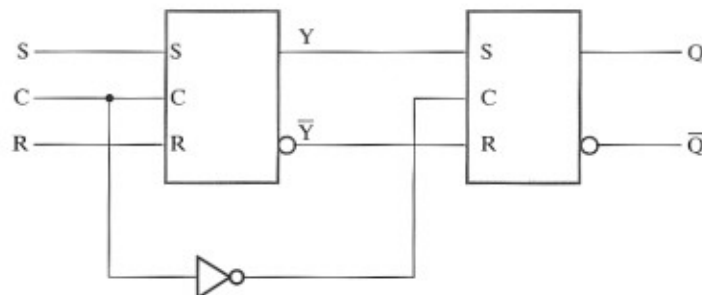
Flip-flop circuits are constructed in such a way as to make them operate properly when they are part of a sequential circuit that employs a single clock.

Note that the problem with the latch is that it is transparent: As soon as an input changes, shortly thereafter the corresponding output changes to match it. This transparency is what allows a change on a latch output to produce additional changes at other latch outputs while the clock pulse is at logic 1. The key to the proper operation of flip-flops is to prevent them from being transparent. In a flip-flop, before an output can change, the path from its inputs to its outputs is broken. So a flip-flop cannot “see” the change of its output or of the outputs of other, like flip-flops at its input during the same clock pulse. Thus, the new state of a flip-flop depends only on the immediately preceding state, and the flip-flops do not go through multiple changes of state.

There are two ways that latches are combined to form a flip-flop. One way is to combine two latches such that (1) the inputs presented to the flip-flop when a clock pulse is present control its state and (2) the state of the flip-flop changes only when a clock pulse is not present. Such a circuit is called a *master-slave* flip-flop. Another way is to produce a flip-flop that triggers only during a signal *transition* from 0 to 1 (or from 1 to 0) on the clock and that is disabled at all other times, including for the duration of the clock pulse. Such a circuit is said to be an *edge-triggered* flip-flop. Next, the implementations of these two flip-flop triggering approaches are presented. It is necessary to consider the SR flip-flop for the master-slave triggering approach since a properly-constructed D flip-flop has the same behavior for both triggering types.

Master-Slave Flip-Flops

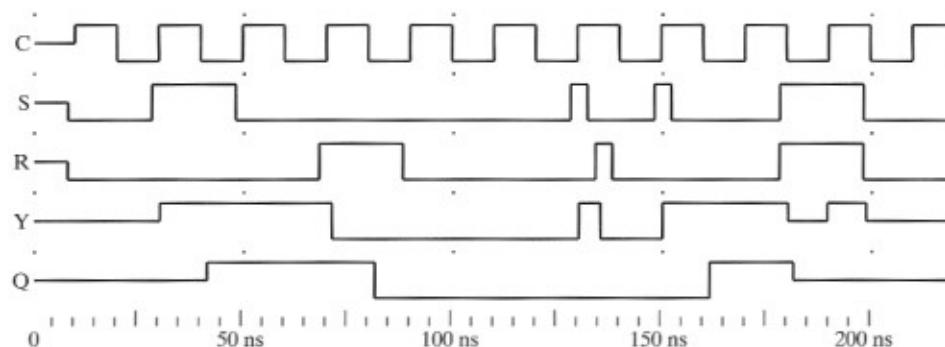
The master-slave *SR* flip-flop, consisting of two latches and an inverter, is shown in Figure 6-10. The symbol with *S*, *C*, and *R* on it is that for the *SR* latch with control input (Figure 6-7), which is referred to here as a clocked *SR* latch. The left clocked *SR* latch in Figure 6-10 is called the master, the right the slave. When the clock input *C* is 0, the output of the inverter is 1. The slave latch is then enabled, and its output *Q* is equal to the master output *Y*. The master latch is disabled, because *C* is 0. When a logic-1 clock pulse is applied, the values on *S* and *R* control the value stored in the master latch *Y*. The slave, however, is disabled as long as the pulse remains at the 1 level, because its *C* input is equal to 0. Any changes in the external



□ **FIGURE 6-10**
SR Master-Slave Flip-Flop

S and R inputs change the master output Y , but cannot affect the slave output Q . When the pulse returns to 0, the master is disabled and is isolated from the S and R inputs. At the same time, the slave is enabled, and the current value of Y is transferred to the output of the flip-flop at Q .

A ModelSim logic simulation illustrating master-slave flip-flop SR behavior is shown in Figure 6-11. Initially, all values are unknown including the clock C . When S and R both go to 0, and the clock goes from 1 to 0, the output of the master, Y and the output of the slave, Q , both remain unknown, since the prior value is effectively being stored. S is at 1 with R at 0 to set the flip-flop in response to the next clock pulse. When C becomes 1, Y sets to 1. When C becomes 0, the slave copies the value of Y setting Q to 1. After S returns to 0, Y and Q remain unchanged, storing the 1 value through the next clock period. Next, R becomes 1. After the clock pulse transition from 0 to 1, the master latch is reset with Y changing to 0. The slave latch is not affected, because its C input is 0. Since the master is an internal circuit, its change of state is not presented at output Q . Even if the inputs S and R change during this interval and the state of the master latch responds by changing, the output of the flip-flop remains in its previous state. When the pulse returns to 0, the information from the master is allowed to pass through to the slave. For the simulation example, the value $Y = 0$ is copied to the slave latch making the external output $Q = 0$. Note that these changes are delayed from the pulse changes by gate delays. Also, the external inputs S and R can change anytime after the clock pulse goes through its negative transition. This is because, as the C input reaches 0, the master is disabled, and S and R have no effect until the next clock pulse. The next sequence of signal changes illustrates the “one’s catching” behavior of the SR master-slave flip-flop. A narrow pulse to 1 occurs on S at the beginning of a clock pulse. The master latch responds to the 1 on S by changing Y to 1. Then S goes to 0 and a narrow 1 pulse occurs on R . The master latch responds to the 1 on R by changing Y back to 0. Since there are no further 1 values on S or R , the master continues to store 0 which is copied to the slave latch, changing Q to 0, in response to the clock changing to 0. Thus, the master latch “caught” both the 1 on S and the 1 on R . Since the 1 on R was caught



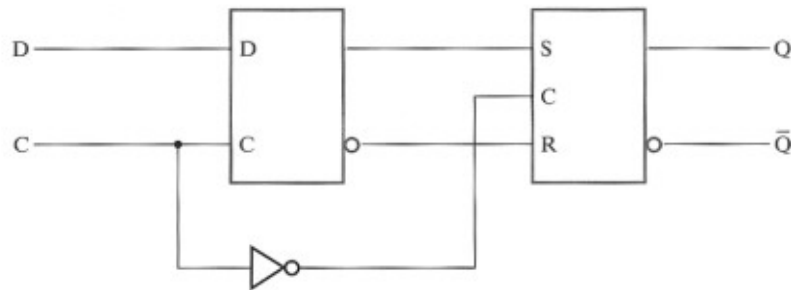
□ **FIGURE 6-11**
Logic Simulation of an SR Master-Slave Flip-Flop

last, the output Q remained at 0. In general, the “correct” response is assumed to be the response to the input values when the clock goes to 0. So, in this case, the response happens to be correct, although more by accident with the changing values in the master. For the next clock pulse, a narrow 1 pulse occurs on S setting the master output Y to 1. The clock then goes to 0 and the value 1 is transferred to the slave latch and appears on Q . In this case, the correct value on Q should be 0 since Q was 0 before the clock pulse and both S and R are 0 just before the clock goes to 0. Since Q equals 1, due to “1’s catching” on S , the flip-flop is in the wrong state. For the final clock pulse of interest, both S and R become 1 before the clock goes to 0. This applies the invalid combination to the master latch making both Y and \bar{Y} equal to 1. When the clock changes to 0, the $\bar{S} \bar{R}$ latch within the master sees its inputs change from (0, 0) to (1, 1), causing the master latch to enter an unknown state which is immediately transferred to the inputs of the slave which also enters an unknown state. This demonstrates that $S = 1, R = 1$ is an invalid input combination for the SR master-slave flip-flop.

Now consider a sequential system containing many master-slave flip-flops, with the outputs of some flip-flops going to inputs of other flip-flops. Assume that the clock pulses to all of the flip-flops are synchronized and occur at the same time. At the beginning of each clock pulse, some of the masters change state, but all the slaves remain in their previous states. This means that the flip-flop slaves are still in their original states, while the flip-flop masters have changed to the new states. After the clock pulse returns to 0, some of the flip-flop slaves change state, but none of the new states have an effect on any of the masters until the next pulse. Thus, the states of flip-flops in a synchronous system can change simultaneously for the same clock pulse, even though outputs of flip-flops are connected to inputs of the same or other flip-flops. This is possible because the inputs affect the state of the flip-flop only while the clock pulse is 1 and the new state appears at the outputs only after the clock pulse has returned to 0, ensuring that the flip-flops are *not* transparent.

For reliable sequential circuit operation, all signals must propagate from the outputs of flip-flops, through the combinational circuit, and back to inputs of master-slave flip-flops, while the clock pulse remains at the logic-0 level. Any changes that occur at the inputs of flip-flops after the clock pulse goes to the logic-1 level, whether intentional or not, affect the flip-flop state and may result in the storage of incorrect values. Suppose that the delay in the combinational circuit is such that S is still changing after the clock pulse has gone to the logic-1 level. Suppose also that, as a consequence, the master is set to 1 by the presence of $S = 1$. When S finally stops changing, it is at 0, indicating that the state of the flip-flop was *not* to be changed from 0. Thus, the 1 value in the master, which will be transferred to the slave, is in error. There are two consequences of this behavior. First, the master-slave flip-flop is also referred to as a *pulse-triggered* flip-flop, since it can respond to input values that cause a change in state and occur anytime during its clock pulse. Second, the circuit must be designed so that combinational circuit delays are short enough to prevent S and R from changing during the clock pulse.

A master-slave D flip-flop can be constructed from the SR master-slave flip-flop by simply replacing the master SR latch with a master D latch. The resulting

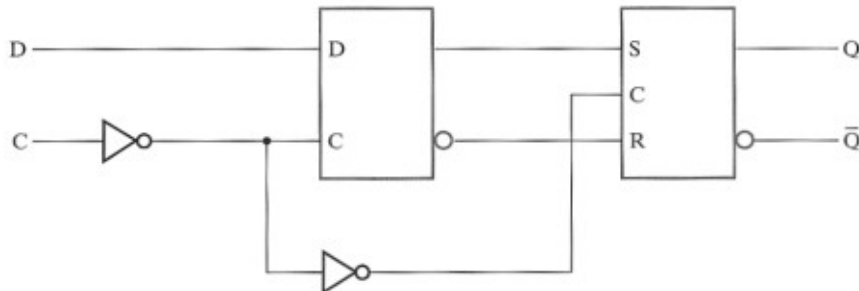


□ **FIGURE 6-12**
Negative Edge-Triggered D Flip-Flop

circuit is shown in Figure 6-12. The resulting circuit changes its value on the negative edge of the clock pulse just as the master-slave SR flip-flop does. However, the *D* type of flip-flop does not demonstrate the usual pulse-triggered behavior. Instead it demonstrates edge-triggered behavior, in this case, negative edge-triggered behavior. Thus, a master-slave *D* flip-flop constructed as shown, is also an edge-triggered flip-flop.

Edge-Triggered Flip-Flop

An *edge-triggered* flip-flop ignores the pulse while it is at a constant level and triggers only during a *transition* of the clock signal. Some edge-triggered flip-flops trigger on the positive edge (0-to-1 transition), whereas others trigger on the negative edge (1-to-0 transition) as illustrated in the previous subsection. The logic diagram of a *D*-type positive-edge-triggered flip-flop to be analyzed in detail here appears in Figure 6-13. This flip-flop takes exactly the form of a master-slave flip-flop, with the master a *D* latch and the slave an *SR* latch or a *D* latch. Also, an inverter is added to the clock input. Because the master latch is a *D* latch, the flip-flop exhibits its edge-triggered rather than master-slave or pulse-triggered behavior. For the clock input equal to 0, the master latch is enabled and transparent and follows the

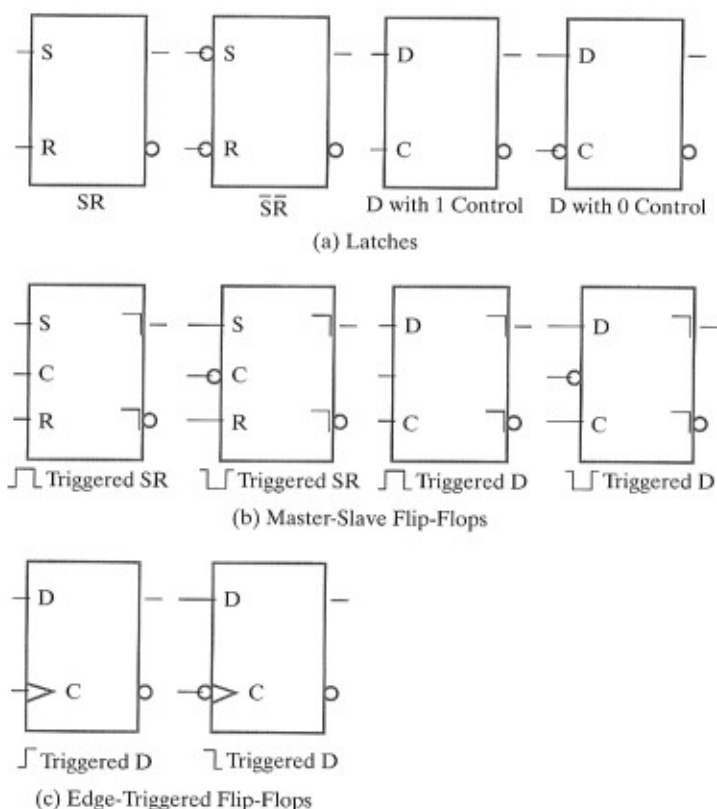


□ **FIGURE 6-13**
Positive Edge-Triggered D Flip-Flop

D input value. The slave latch is disabled and holds the state of the flip-flop fixed. When the positive edge occurs, the clock input changes to 1. This disables the master latch so that its value is fixed and enables the slave latch so that it copies the state of the master latch. The state of the master latch to be copied is the state that is present at the positive edge of the clock. Thus, the behavior appears to be edge-triggered. With the clock input equal to 1, the master latch is disabled and cannot change, so the state of both the master and the slave remain unchanged. Finally, when the clock input changes from 1 to 0, the master is enabled and begins following the D value. But during the 1-to-0 transition, the slave is disabled before any change in the master can reach it. Thus, the value stored in the slave remains unchanged during this transition. An alternative implementation is given in Problem 6-3 at the end of the chapter.

Standard Graphics Symbols

The standard graphics symbols for the different types of latches and flip-flops are shown in Figure 6-14. A flip-flop or latch is designated by a rectangular block with



□ **FIGURE 6-14**
Standard Graphics Symbols for Latches and Flip-Flops

inputs on the left and outputs on the right. One output designates the normal state of the flip-flop, and the other, with a bubble, designates the complement output. The graphics symbol for the SR latch or SR flip-flop has inputs S and R indicated inside the block. In the case of the $\overline{S}\overline{R}$ latch, bubbles are added to the inputs to indicate that setting and resetting occur for 0-level inputs. The graphics symbol for the D latch or D flip-flop has inputs D and C indicated inside the block.

Below each symbol, a descriptive title, which is not part of the symbol, is given. In the titles, \sqcap denotes a positive pulse, \sqcup a negative pulse, \lceil a positive edge, and \rfloor a negative edge.

Triggering by the 0 level rather than the 1 level is denoted on the latch symbols by adding a bubble at the triggering input. The master-slave is a pulse-triggered flip-flop and is indicated as such with a right-angle symbol called a *postponed output indicator* in front of the outputs. This symbol shows that the output signal changes at the end of the pulse. To denote that the master-slave flip-flop will respond to a negative pulse (i.e., a pulse to 0 with the inactive clock value at 1), a bubble is placed on the C input. To denote that the edge-triggered flip-flop responds to an edge, an arrowhead-like symbol in front of the letter C designates a *dynamic input*. This *dynamic indicator* symbol denotes the fact that the flip-flop responds to edge transitions of the input clock pulses. A bubble outside the block adjacent to the dynamic indicator designates a negative-edge transition for triggering the circuit. The absence of a bubble designates a positive-edge transition for triggering.

Often, all of the flip-flops used in a circuit are of the same triggering type, such as positive-edge triggered. All of the flip-flops will then change in relation to the same clocking event. When using flip-flops having different triggering in the same sequential circuit, one may still wish to have all of the flip-flop outputs change relative to the same clocking event. Those flip-flops that behave in a manner opposite from the adopted polarity transition can be changed by the addition of inverters to their clock inputs. A preferred procedure is to provide both positive and negative pulses from the master clock generator that are carefully aligned. We apply positive pulses to positive-pulse-triggered (master-slave) and negative-edge-triggered flip-flops and negative pulses to negative-pulse-triggered (master-slave) and positive-edge-triggered flip-flops. In this way, all flip-flop outputs will change at the same time. Finally, to prevent specific timing problems, some designers use flip-flops having different triggering (i.e., both positive and negative edge-triggered flip-flops) with a single clock. In these cases, flip-flop outputs are purposely made to change at different times.

In this text, it is assumed that all flip-flops are of the positive-edge-triggered type, unless otherwise indicated. This provides a uniform graphics symbol for the flip-flops and consistent timing diagrams.

Note that there is no input to the D flip-flop that produces a “no change” condition. This condition can be accomplished either by disabling the clock pulses on the C input or by leaving the clock pulses undisturbed and connecting the output back into the D input using a multiplexer when the state of the flip-flop must remain the same. The technique that disables clock pulses is referred to as *clock gating*. This technique typically uses fewer gates and saves power, but is often avoided because the gated clock pulses into the flip-flops are delayed. The delay,

called *clock skew*, causes gated clock and nongated clock flip-flops to change at different times. This can make the circuit unreliable, since the outputs of some flip-flops may reach others while their inputs are still affecting their state.

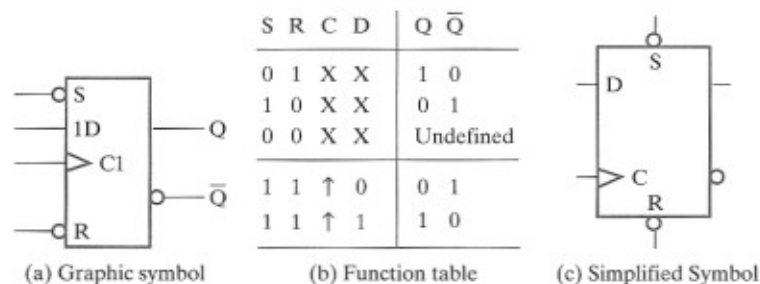
Direct Inputs

Flip-flops often provide special inputs for setting and resetting them asynchronously (i.e., independently of the clock input C). The inputs that asynchronously set the flip-flop are called *direct set*, or *preset*. The inputs that asynchronously reset the flip-flop are called *direct reset*, or *clear*. Application of a logic 1 (or a logic 0 if a bubble is present) to these inputs affects the flip-flop output without the use of the clock. When power is turned on in a digital system, the states of its flip-flops can be anything. The direct inputs are useful for bringing flip-flops in a digital system to an initial state prior to the normal clocked operation.

The IEEE standard graphics symbol for a positive-edge-triggered D flip-flop with direct set and direct reset is shown in Figure 6-15(a). The notations, $C1$ and $1D$, illustrate control dependency. An input labeled Cn , where n is any number, controls all the other inputs starting with the number n . In the figure, $C1$ controls input $1D$. S and R have no 1 in front of them, and therefore, they are not controlled by the clock at $C1$. The S and R inputs have circles on the input lines to indicate that they are active at the logic-0 level (i.e., a 0 applied will result in the set or reset action).

The function table in Figure 6-15(b) specifies the operation of the circuit. The first three rows in the table specify the operation of the direct inputs S and R . These inputs behave like NAND $\bar{S}\bar{R}$ latch inputs (see Figure 6-6), operating independently of the clock, and are therefore asynchronous inputs. The last two rows in the function table specify the clocked operation for values of D . The clock at C is shown with an upward arrow to indicate that the flip-flop is a positive-edge-triggered type. The D input effects are controlled by the clock in the usual manner.

Figure 6-15(c) shows a less formal symbol for the positive-edge-triggered flip-flop with direct set and reset. The positioning of S and R at the top and bottom of the symbol rather than on the left edge implies that resulting output changes are not controlled by the clock C .

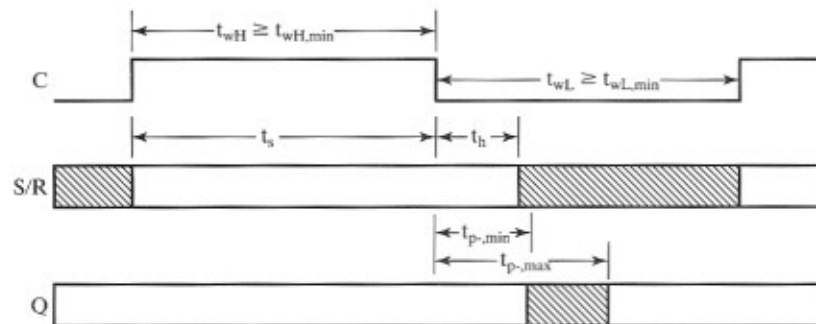


□ **FIGURE 6-15**
 D Flip-Flop with Direct Set and Reset

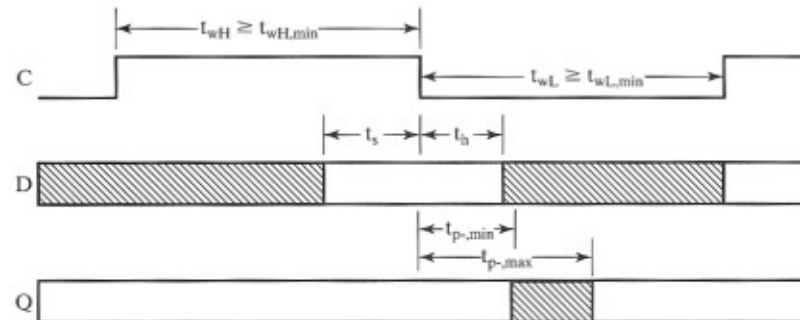
Flip-Flop Timing

There are timing parameters associated with the operation of both pulse-triggered and edge-triggered flip-flops. These parameters are illustrated for a master-slave *SR* flip-flop and for a negative-edge-triggered *D* flip-flop in Figure 6-16. The parameters for the positive-edge-triggered *D* flip-flop are the same except that they are referenced to the positive clock edge rather than the negative clock edge.

The timing of the response of a flip-flop to its inputs and clock *C* must be taken into account when using the flip-flops. For both flip-flops, there is a minimum time called the *setup time*, t_s , for which the *S* and *R* or *D* inputs must be maintained at a constant value prior to the occurrence of the clock transition that causes the output to change. Otherwise, the master could be changed erroneously in the case of the master-slave flip-flop or be at an intermediate value at the time the slave copies it in the case of the edge-triggered flip-flop. Similarly, there is a minimum time called the *hold time*, t_h , for which the *S* and *R* or *D* inputs must not change after the application of the clock transition that causes the output to change. Otherwise, the master might respond to the input change and be changing at the time the slave latch copies it. In addition, there is a minimum *clock pulse width* t_w , to



(a) Pulse-triggered (positive pulse)



(b) Edge-triggered (negative edge)

□ **FIGURE 6-16**
Flip-flop Timing Parameters

insure that the master has time enough to capture the input values correctly. Among these parameters, the one that differs most between the pulse-triggered and edge-triggered flip-flops is the setup time as shown in Figure 6-16. The pulse-triggered flip-flop has its setup time equal to the clock pulse width, whereas the setup time for the edge-triggered flip-flop can be much smaller than the clock pulse width. As a consequence, edge-triggering tends to provide faster designs since the flip-flop inputs can change later with respect to the upcoming triggering clock edge.

The *propagation delay times*, t_{PHL} , t_{PLH} , or t_{pd} of the flip-flops are defined as the interval between the triggering clock edge and the stabilization of the output to a new value. These times are defined in the same fashion as those for an inverter except that the values are measured from the triggering clock edge rather than the inverter input. In Figure 6-16, all of these parameters are denoted by t_p , and are given minimum and maximum values. Since the changes of the flip-flop outputs are to be separated from the control by the flip-flop inputs, the minimum propagation delay time should be longer than the hold time for correct operation. These and other parameters are specified in manufacturers' data books for specific integrated circuit products.

Similar timing parameters can be defined for latches and direct inputs, with additional propagation delays needed to model the transparent behavior of latches.

6-4 SEQUENTIAL CIRCUIT ANALYSIS

The behavior of a sequential circuit is determined from the inputs, outputs, and present state of the circuit. The outputs and the next state are a function of the inputs and the present state. The analysis of a sequential circuit consists of obtaining a suitable description that demonstrates the time sequence of inputs, outputs, and states.

A logic diagram is recognized as a synchronous sequential circuit if it includes flip-flops with the clock inputs driven directly or indirectly by a clock signal and if the direct sets and resets are unused during the normal functioning of the circuit. The flip-flops may be of any type, and the logic diagram may or may not include combinational gates. In this section, an algebraic representation for specifying the logic diagram of a sequential circuit is given. A state table and state diagram are presented that describe the behavior of the circuit. Specific examples will be used throughout the discussion to illustrate the various procedures.

Input Equations

The logic diagram of a sequential circuit consists of flip-flops and, usually, combinational gates. The knowledge of the type of flip-flops used and a list of Boolean functions for the combinational circuit provide all the information needed to draw the logic diagram of the sequential circuit. The part of the combinational circuit that generates the signals for the inputs of flip-flops can be described by a set of Boolean functions called *flip-flop input equations*. We will adopt the convention of using the flip-flop input symbol to denote the flip-flop input equation variable and using the name of the flip-flop output as the subscript for the variable. From this

example, it becomes apparent that a flip-flop input equation is a Boolean expression for a combinational circuit. The subscripted symbol is an output variable of the combinational circuit. This output is always connected to the input of a flip-flop—thus the name “flip-flop input equation.”

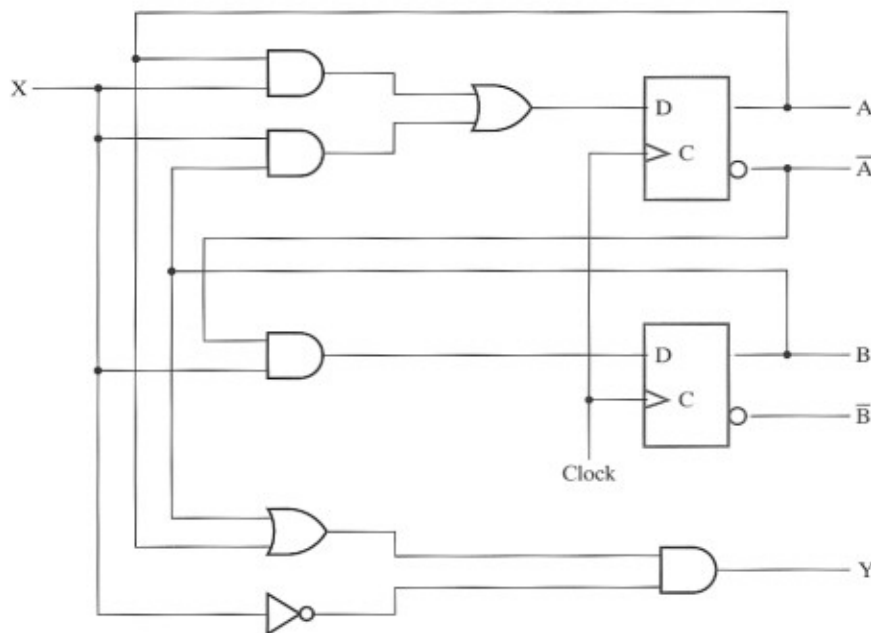
The flip-flop input equations constitute a convenient algebraic expression for specifying the logic diagram of a sequential circuit. They imply the type of flip-flop from the letter symbol, and they fully specify the combinational circuit that drives the flip-flops. Time is not included explicitly in these equations, but is implied from the clock at the C input of the flip-flops. An example of a sequential circuit is given in Figure 6-17. The circuit has two D -type flip-flops, an input X , and an output Y . It can be specified by the following equations:

$$D_A = AX + BX$$

$$D_B = \bar{A}X$$

$$Y = (A + B)\bar{X}$$

The first two equations are for flip-flop inputs, and the third equation specifies the output Y . Note that the input equations use the symbol D , which is the same as the input symbol of the flip-flops. The subscripts A and B designate the outputs of the respective flip-flops.



□ **FIGURE 6-17**
Example of a Sequential Circuit

State Table

The functional relationships among the inputs, outputs, and flip-flop states of a sequential circuit can be enumerated in a *state table*. The state table for the circuit of Figure 6-17 is shown in Table 6-1. The table consists of four sections, labeled *present state*, *input*, *next state*, and *output*. The present-state section shows the states of flip-flops A and B at any given time t . The input section gives each value of X for each possible present state. Note that for each possible input combination, each of the present states is repeated. The next-state section shows the states of the flip-flops one clock period later, at time $t + 1$. The output section gives the value of Y at time t for each combination of present state and input.

The derivation of a state table consists of first listing all possible binary combinations of present state and inputs. In Table 6-1, there are eight binary combinations, from 000 to 111. The next-state values are then determined from the logic diagram or from the flip-flop input equations. For a D flip-flop, the relationship $A(t + 1) = D_A(t)$ holds. This means that the next state of flip-flop A is equal to the present value of its input D . The value of the D input is specified in the flip-flop input equation as a function of the present state of A and B and input X . Therefore, the next state of flip-flop A must satisfy the equation

$$A(t + 1) = D_A = AX + BX$$

The next-state section in the state table under column A has three 1's where the present state and input value satisfy the conditions $(A, X) = 11$ or $(B, X) = 11$. Similarly, the next state of flip-flop B is derived from the input equation

$$B(t + 1) = D_B = \bar{A}X$$

and is equal to 1 when the present state of A is 0 and input X is equal to 1. The output column is derived from the output equation

$$Y = A\bar{X} + B\bar{X}$$

□ **TABLE 6-1**
State Table for Circuit of Figure 6-17

Present State		Input	Next State		Output
A	B	X	A	B	Y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0

□ **TABLE 6-2**
Two-Dimensional State Table for the Circuit in Figure 6-17

Present state		Next state				Output	
		X = 0		X = 1		X = 0	X = 1
A	B	A	B	A	B	Y	Y
0	0	0	0	0	1	0	0
0	1	0	0	1	1	1	0
1	0	0	0	1	0	1	0
1	1	0	0	1	0	1	0

The state table of any sequential circuit with D -type flip-flops is obtained in this way. In general, a sequential circuit with m flip-flops and n inputs needs 2^{m+n} rows in the state table. The binary numbers from 0 through $2^{m+n} - 1$ are listed in the combined present-state and input columns. The next-state section has m columns, one for each flip-flop. The binary values for the next state are derived directly from the D flip-flop input equations. The output section has as many columns as there are output variables. Its binary values are derived from the circuit or from the Boolean functions in the same manner as in a truth table.

Table 6-1 is one-dimensional in the sense that the present state and input combinations are combined into a single column of combinations. A two-dimensional state table having the present state tabulated in the left column and the inputs tabulated across the top row is also frequently used. The next-state entries are made in each cell of the table for the present-state and input combination corresponding to the location of the cell. A similar two-dimensional table is used for the outputs if they depend upon the inputs. Such a state table is shown in Table 6-2. Sequential circuits in which the outputs depend on the inputs, as well as on the states, are referred to as *Mealy model* circuits. Otherwise, if the outputs depend only on the states, then a one-dimensional column suffices. In this case, the circuits are referred to as *Moore model* circuits. Each model is named after its originator.

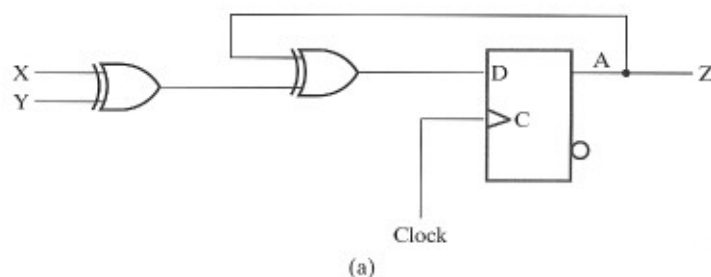
As an example of a Moore model circuit, suppose we want to obtain the logic diagram and state table of a sequential circuit that is specified by the flip-flop input equation

$$D_A = A \oplus X \oplus Y$$

and output equation

$$Z = A$$

The D_A symbol implies a D -type flip-flop with output designated by the letter A . The X and Y variables are taken as inputs and Z as the output. The logic diagram and state table for this circuit are shown in Figure 6-18. The state table has one column for the present state and one column for the inputs. The next state and output are also in single columns. The next state is derived from the flip-flop input



Present state	Inputs		Next state	Output
A	X	Y	A	Z
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	0
1	0	0	1	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

(b) State table

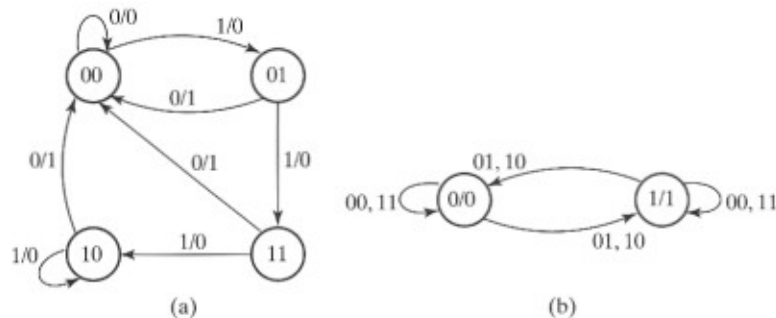
□ FIGURE 6-18

Logic Diagram and State Table for $D_A = A \oplus X \oplus Y$

equation which specifies an odd function. (See Section 2-8.) The output column is simply a copy of the column for the present-state variable A .

State Diagram

The information available in a state table may be represented graphically in the form of a state diagram. In this type of diagram, a state is represented by a circle, and transitions between states are indicated by directed lines connecting the circles. Examples of state diagrams are given in Figure 6-19. Figure 6-19(a) shows the state diagram for the sequential circuit in Figure 6-17 and its state table in Table 6-1. The state diagram provides the same information as the state table and is obtained directly from it. The binary number inside each circle identifies the state of the flip-flops. For Mealy model circuits, the directed lines are labeled with two binary numbers separated by a slash. The input value during the present state precedes the slash, and the value following the slash gives the output value during the present state with the given input applied. For example, the directed line from state 00 to state 01 is labeled 1/0, meaning that when the sequential circuit is in the present state 00 and the input is 1, the output is 0. After the next clock transition, the circuit goes to the next state, 01. If the input changes to 0, then the output becomes 1, but if the input remains at 1, the output stays at 0. This information is obtained from the state diagram along the two directed



□ **FIGURE 6-19**
State Diagrams

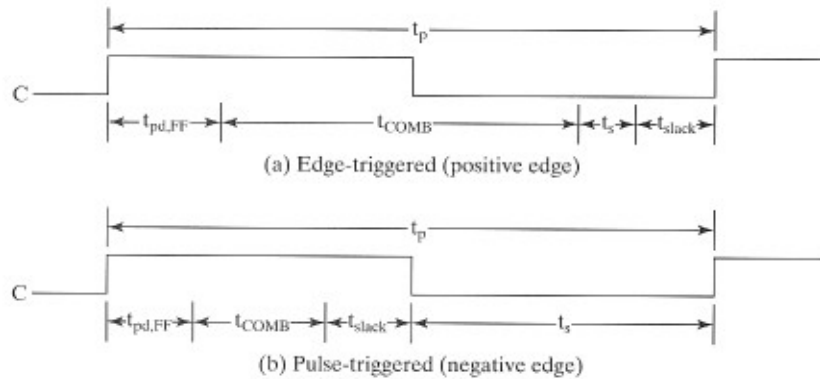
lines emanating from the circle with state 01. A directed line connecting a circle with itself indicates that no change of state occurs.

The state diagram of Figure 6-19(b) is for the sequential circuit of Figure 6-18. Here, only one flip-flop with two states is needed. There are two binary inputs, and the output depends only on the state of the flip-flop. For such a Moore model circuit, the slash on the directed lines is not included, since the outputs depend only on the state and not on the input values. Instead, the output is included under a slash below the state in a circle. There are two input conditions for each state transition in the diagram, and they are separated by a comma. When there are two input variables, each state may have up to four directed lines coming out of the corresponding circle, depending upon the number of states and the next state for each binary combination of the input values.

There is no difference between a state table and a state diagram, except for their manner of representation. The state table is easier to derive from a given logic diagram and input equations. The state diagram follows directly from the state table. The state diagram gives a pictorial view of state transitions and is the form more suitable for human interpretation of the operation of the circuit. For example, the state diagram of Figure 6-19(a) clearly shows that, starting at state 00, the output is 0 as long as the input stays at 1. The first 0 input after a string of 1's gives an output of 1 and sends the circuit back to the initial state of 00. The state diagram of Figure 6-19(b) shows that the circuit stays at a given state as long as the two inputs have the same value (00 or 11). There is a state transition between the two states only when the two inputs are different (01 or 10).

Sequential Circuit Timing

In addition to analyzing the function of a circuit, it is also important to analyze its performance in terms of the *maximum input-to-output delay* and the *maximum clock frequency*, f_{\max} at which it can operate. First of all, the clock frequency is just the inverse of the clock period t_p shown in Figure 6-20. So, the maximum allowable clock frequency corresponds to the minimum allowable clock period t_p . To determine how small we can make the clock period, we need to determine the longest



□ **FIGURE 6-20**
Sequential Circuit Timing Parameters

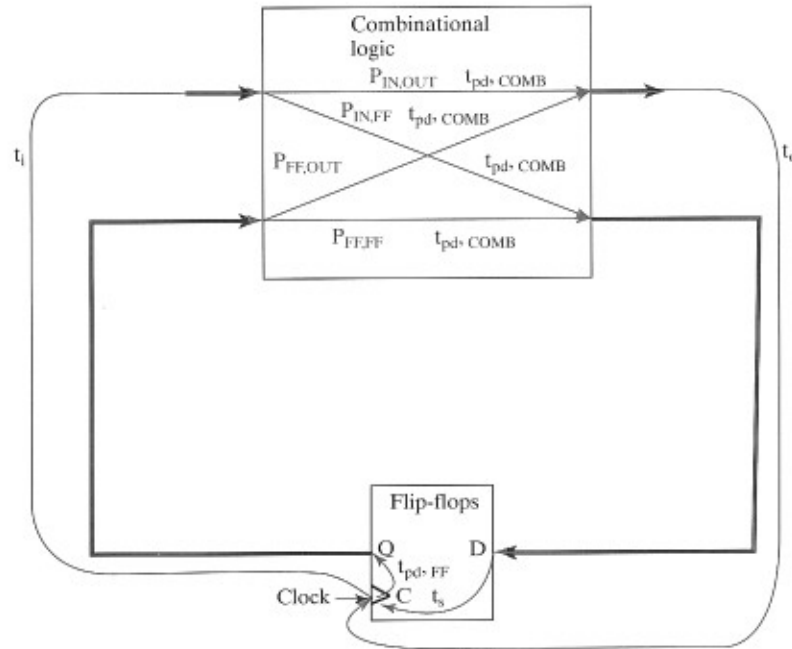
delay from the triggering edge of the clock to the next triggering edge of the clock. These delays are measured on all such paths in the circuit down which changing signals propagate. Each of these path delays has three components: (1) a flip-flop propagation delay, $t_{pd,FF}$ (2) a combinational logic delay through the chain of gates along the path, $t_{pd,COMB}$, and (3) a flip-flop setup time, t_s . As a signal change propagates down the path, it is delayed successively by an amount equal to each of these delays. Note that we have used t_{pd} , instead of the more detailed values, t_{PLH} and t_{PHL} , for both the flip-flops and combinational logic gates to simplify the delay calculations. Figure 6-20 summarizes the delay picture for both the edge-triggered and pulse-triggered flip-flops.

After a positive edge on a clock, if a flip-flop is to change, its output changes at time $t_{pd,FF}$ after the clock edge. This change enters the combinational logic path and must propagate down the path to a flip-flop input. This requires an additional time, $t_{pd,COMB}$, for the signal change to reach the second flip-flop. Finally, before the next positive clock edge, this change must be held on the flip-flop input for setup time t_s . This path, $P_{FF,FF}$ as well as other possible paths are illustrated in Figure 6-21. For paths $P_{IN,FF}$ driven by primary inputs, $t_{pd,FF}$ is replaced by t_i , which is the latest time that the input changes after the positive clock edge. For a path $P_{FF,OUT}$ driving primary outputs, t_s is replaced by t_o , which is the latest time that the output is permitted to change prior to the next clock edge. Finally, in a Mealy model circuit, combinational paths from input to output, $P_{IN,OUT}$, that use both t_i and t_o can appear. Each path has a slack time, t_{slack} , the extra time allowed in the clock period beyond that required by the path. From Figure 6-21, the following equation for a path of type $P_{FF,FF}$ results:

$$t_p = t_{slack} + (t_{pd,FF} + t_{pd,COMB} + t_s)$$

In order to guarantee that a changing value is captured by the receiving flip-flop, t_{slack} must be greater than or equal to zero for all of the paths. This requires that

$$t_p \geq \max (t_{pd,FF} + t_{pd,COMB} + t_s) = t_{p,min}$$



□ **FIGURE 6-21**
Sequential Circuit Timing Paths

where the maximum is taken over all paths down which signals propagate from flip-flop to flip-flop. The next example presents representative calculations for paths $P_{FF,FF}$

EXAMPLE 6-1 Clock Period and Frequency Calculations

Suppose that all flip-flops used are the same and have $t_{pd} = 0.2$ ns (nanosecond = 10^{-9} seconds) and $t_s = 0.1$ ns. Then the longest path beginning and ending with a flip-flop will be the path with the largest $t_{pd, COMB}$. Further, suppose that the largest $t_{pd, COMB}$ is 1.3 ns and that t_p has been set to 1.5 ns. From the previous equation for t_p , we can write

$$1.5 \text{ ns} = t_{\text{slack}} + 0.2 + 1.3 + 0.1 = t_{\text{slack}} + 1.6 \text{ ns}$$

Solving, we have $t_{\text{slack}} = -0.1$ ns, so this value of t_p is too small. In order for t_{slack} to be greater than or equal to zero for the longest path, $t_p \geq t_{p, \text{min}} = 1.6$ ns. The maximum frequency $f_{\text{max}} = 1 / 1.6 \text{ ns} = 625 \text{ MHz}$ (megahertz = 10^6 cycles per second). We note that, if t_p is too large to meet the circuit specifications, we must either employ faster logic cells or change the circuit design to reduce the problematic path delays through the circuit while still performing the desired function. ■

It is interesting to note that the hold time for a flip-flop, t_h , does not appear in the clock period equation. It relates to another timing constraint equation dealing with one or both of two specific situations. In one case, output changes arrive at the inputs of one or more flip-flops too soon. In the other case, the clock signals reaching one or more flip-flops are somehow delayed, a condition referred to as *clock skew*. Clock skew also can affect the maximum clock frequency.

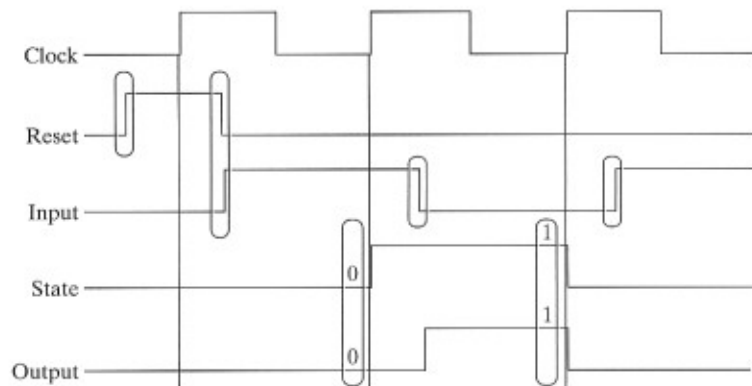
Simulation

Sequential circuit simulation involves issues not present in combinational circuits. First of all, rather than a set of input patterns for which the order of application is immaterial, the patterns must be applied in a sequence. This sequence includes timely application of input patterns as well as clock pulses. Second, there must be some means to place the circuit in a known state. Realistically, initialization to a known state is accomplished by application of an initialization subsequence at the beginning of the simulation. In the simplest case, this subsequence is a reset signal. For flip-flops lacking a circuit reset (or set), a longer sequence typically consisting of an initial reset followed by a sequence of normal input patterns is required. A simulator may also have a means of setting the initial state which is useful to avoid long sequences that may be needed to get to an initial state. Aside from getting to an initial state, a third issue is observing the state to verify correctness. In some circuits, application of an additional sequence of inputs is required to determine the state of the circuit at a given point. The simplest alternative is to set up the simulation so that the state of the circuit can be observed directly; the approach to doing this varies depending on the simulator and whether or not the circuit contains hierarchy. A crude approach that works with all simulators is to add a circuit output with a path from each state variable signal.

A final issue to be dealt with in more detail is the timing of application of inputs and observation of outputs relative to the active clock edge. Initially, we discuss the timing for *functional simulation* having as its objective determination or verification of the function of the circuit. In functional simulation, components of the circuit have no delay or a very small delay. Much more complex is *timing simulation* in which the circuit elements have realistic delays and verification of the proper operation of the circuit in terms of timing is the simulation objective.

Some simulators, by default, use a very small component delay for functional simulation so that the order of changes in signals can be observed provided that the time scale used for display is small enough. Suppose that the component delays and the setup and hold times for flip-flops are all 0.1 ns for such a simulation and that the longest delay from positive clock edge to positive clock edge is 1.2 ns in your circuit. If you happen to use a clock period of 1.0 ns for your simulation, when the result depends on the longest delay, the simulation results will be in error! So for functional simulation with such a simulator, either a longer clock period should be chosen for the simulation or the default delay needs to be changed by the user to a smaller value.

In addition to the clock period, the time of application of inputs relative to the positive clock edge is important. For functional simulation, to allow for any small, default component delays, the inputs for a given clock cycle should be



□ **FIGURE 6-22**
Simulation Timing

changed well before the positive clock edge, preferably early in the clock cycle while the clock is still at a 1 value. This is also an appropriate time to change the reset signal values to insure that the reset signal is controlling the state rather than the clock edge or a meaningless combination of clock and reset.

A final issue is the time at which to examine a simulation result in functional simulation. At the very latest, the state variable values and outputs should be at their final values just before the positive clock edge. Although it may be possible to observe the values at other locations, this location provides a foolproof observation time for functional simulation.

The ideas just presented are summarized in Figure 6-22. Input changes in Reset and Input, encircled in blue, occur at about the 25% point in the clock cycle. Signal values on State and Output, as well as on Input and Reset, all encircled in blue and listed, are observed just before the 100% point in the clock cycle.

6-5 SEQUENTIAL CIRCUIT DESIGN

The design of clocked sequential circuits starts from a set of specifications and culminates in a logic diagram or a list of Boolean functions from which the logic diagram can be obtained. In contrast to a combinational circuit, which is fully specified by a truth table, a sequential circuit requires a state table for its specification. Thus, the first step in the design of a sequential circuit is to obtain a state table or an equivalent representation such as a state diagram.

A synchronous sequential circuit is made up of flip-flops and combinational gates. The design of the circuit consists of choosing the flip-flops and finding a combinational circuit structure which, together with the flip-flops, produces a circuit that fulfills the stated specifications. The minimum number of flip-flops is determined by the number of states in the circuit; n flip-flops can represent up to 2^n binary states. The combinational circuit is derived from the state table by evaluating the flip-flop input equations and output equations. In fact, once the type and

number of flip-flops are determined, the design process transforms a sequential circuit problem into a combinational circuit problem. In this way, the techniques of combinational circuit design can be applied.

Design Procedure

The following procedure for the design of sequential circuits is similar to that for combinational circuits but has some additional steps:

1. **Specification:** Write a specification for the circuit, if not already available.
2. **Formulation:** Obtain either a state diagram or a state table from the statement of the problem.
3. **State Assignment:** If only a state diagram is available from step 1, obtain the state table. Assign binary codes to the states in the table.
4. **Flip-Flop Input Equation Determination:** Select the flip-flop type or types. Derive the flip-flop input equations from the next-state entries in the encoded state table.
5. **Output Equation Determination:** Derive output equations from the output entries in the state table.
6. **Optimization:** Optimize the flip-flop input equations and output equations.
7. **Technology Mapping:** Draw a logic diagram of the circuit using flip-flops, ANDs, ORs, and inverters. Transform the logic diagram to a new diagram using the available flip-flop and gate technology.
8. **Verification:** Verify the correctness of the final design.

For convenience, we usually omit the technology mapping in step 7 and use only flip-flops, AND gates, OR gates, and inverters in the schematic.

Finding State Diagrams and State Tables

The specification for a circuit is often in the form of a verbal description of the behavior of the circuit. This description needs to be interpreted in order to find a state diagram or state table in the formulation step of the design procedure. This is often the most creative part of the design procedure, with many of the subsequent steps performed automatically by computer-based tools.

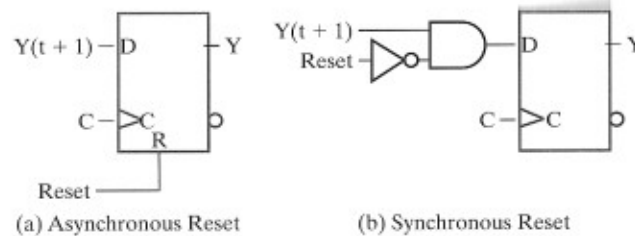
Fundamental to the formulation of state diagrams and tables is an intuitive understanding of the concept of a state. A state is used to “remember” something about the history of input combinations applied to the circuit at either triggering clock edges or during triggering pulses. In some cases, the states may literally store input values retaining a complete history of the sequence appearing on the inputs. In most cases, however, a state is an *abstraction* of the sequence of input combinations at the triggering points. For example, a given state S_1 may represent the fact that among the sequence of values applied to a single bit input X , “the value 1 has appeared on X for the last three consecutive clock edges.” Thus, the circuit would be in state S_1 after sequences ... 00111 or ... 0101111, but would not be in state S_1

after sequences ... 00011 or ... 011100. A state S_2 might represent the fact that the sequence of 2-bit input combinations applied are “in order 00, 01, 11, 10 with any number of consecutive repetitions of each combination permitted and 10 as the most recently applied combination.” The circuit would be in state S_2 for the following example sequences: 00, 00, 01, 01, 01, 11, 10, 10 or 00, 01, 11, 11, 11, 10. The circuit would not be in state S_2 for sequences: 00, 11, 10, 10 or 00, 00, 01, 01, 11, 11. In formulating a state diagram or state table it is useful to write down the abstraction represented by each state. In some cases, it may be easier to describe the abstraction by referring to values that have occurred on the outputs as well as on the inputs. For example, state S_3 might represent the abstraction that “the output bit Z_2 is 1, and the input combination has bit X_2 at 0.” In this case, Z_2 equal to 1 might uniquely represent a complex set of past sequences of input combinations that would be more difficult to describe in detail.

As one formulates a state table or state diagram, new states are added. There is potential for the set of states to become unnecessarily large or potentially even infinite in size! Instead of adding a new state for every current state and possible applied input combination, it is essential that states be reused as next states to prevent uncontrolled state growth as outlined above. The mechanism for doing this is a knowledge of the abstraction that each state represents. To illustrate, consider state S_1 defined previously as an abstraction “the value 1 has appeared at the last three consecutive clock edges.” If S_1 has been entered due to the sequence ... 00111 and the next input is a 1, giving sequence ... 001111, is a new state needed or can the next state be S_1 ? By examining the new sequence, we see that the last three input values are 1's, which matches the abstraction defined for state S_1 . So, state S_1 can be used as the next state for current state S_1 and input value 1, avoiding the definition of a new state.

Since one does not know the state of the flip-flops when the power is first turned on, it is customary to provide a *master reset* signal to initialize the state of the flip-flops in at least some component circuits. This avoids starting these circuits in an unknown state. Typically, the master reset signal is applied to the flip-flops at their asynchronous (direct) inputs (see Figure 6-23) before clocked operation starts. In most cases flip-flops are reset to 0, but some may be set to 1, depending upon the initial state desired. If there is a large number of flip-flops, based upon the circuit operation, only a portion of them may be initialized.

When the power in a digital system is first turned on, the state of the flip-flops is unknown. It is possible to apply an input sequence with the circuit in an unknown state, but that sequence must be able to bring a portion of the circuit to a known state before meaningful outputs can be expected. In fact, many of the larger sequential circuits we design in subsequent chapters will be of this type. In this chapter, however, the circuits that we design must have a known *initial state*, and further, a hardware mechanism must be provided to get the circuit from any unknown state into this state. This mechanism is a *reset* or *master reset* signal. Regardless of all other inputs applied to the circuit, the reset places the circuit in its initial state. In fact, the initial state is often called the *reset state*. The reset signal is usually activated automatically when the circuit is powered up. In addition, it may be activated electronically or by pushing a reset button.



□ **FIGURE 6-23**
Asynchronous and Synchronous Reset for D Flip-flops

The reset may be asynchronous, taking place without clock triggering. In this case, the reset is applied to the direct inputs on the circuit flip-flops, as shown in Figure 6-23 (a). This design assigns 00...0 to the initial state of the flip-flops to be reset. If an initial state with a different code is desired, then the *Reset* signal can be selectively connected to direct set inputs instead of direct reset inputs. It is important to note that these inputs should not be used in the normal synchronous circuit design process. Instead, they are reserved only for an asynchronous reset that returns the system, of which the circuit is a component, to an initial state. Using these direct inputs as a part of the synchronous circuit design violates the fundamental synchronous circuit definition, since it permits a flip-flop state to change asynchronously within direct clock triggering.

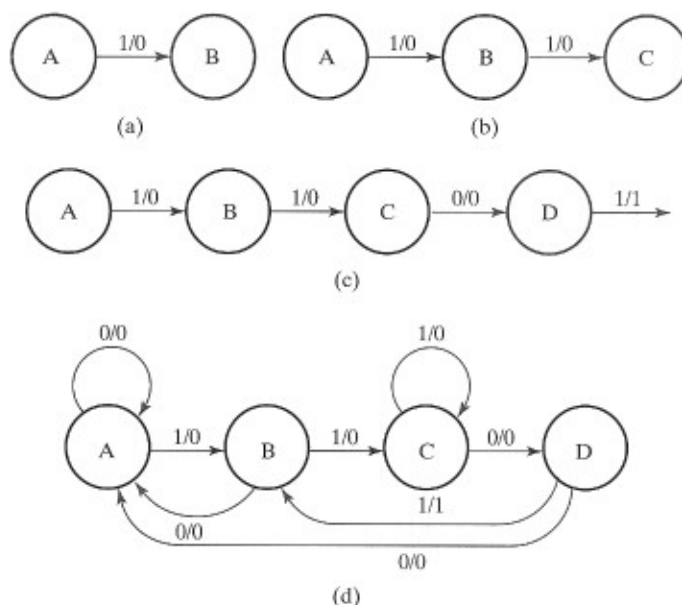
Alternatively, the reset may be synchronous and require a clock triggering event to occur. The reset must be incorporated into the synchronous design of the circuit. A simple approach to synchronous reset for D flip-flops, without formally including the reset bit in the input combinations, is to add the AND gate shown in Figure 6-23 (b) after doing the normal circuit design. This design also assigns 00 ... 0 to the initial state. If a different initial state code is desired, then OR gates with *Reset* as an input can selectively replace the AND gates with inverted *Reset*.

To illustrate the formulation process, two examples follow, each resulting in a different style of state diagram.

EXAMPLE 6-2 Finding a State Diagram for a Sequence Recognizer

The first example is a circuit that recognizes the occurrence of a particular sequence of bits, regardless of where it occurs in a longer sequence. This “sequence-recognizer” has one input *X* and one output *Z*. It has *Reset* applied to the direct reset inputs on its flip-flops to initialize the state of the circuit to all zeros. The circuit is to recognize the occurrence of the sequence of bits 1101 on *X* by making *Z* equal to 1 when the previous three inputs to the circuit were 110 and current input is a 1. Otherwise, *Z* equals 0.

The first step in the formulation process is to determine whether the state diagram or table must be a Mealy model or Moore model circuit. The portion of the preceding specification that says “... making *Z* equal to 1 when the previous three inputs to the circuit are 110 and the current input is a 1” implies that the output is determined from not only the current state, but also the current input. As a



□ **FIGURE 6-24**
Construction of a State Diagram for Example 6.2

consequence, a Mealy model circuit with the output dependent on both state and inputs is required.

Recall that a key factor in the formulation of any state diagram is to recognize that states are used to “remember” something about the history of the inputs. For example, for the sequence 1101, to be able to produce the output value 1 coincident with the final 1 in the sequence, the circuit must be in a state that “remembers” that the previous three inputs were 110. With this concept in mind, we begin to formulate the state diagram by defining an arbitrary initial state A as the reset state and the state in which “none of the sequence to be recognized has occurred.” If a 1 occurs on the input, since 1 is the first bit in the sequence, this event must be “remembered,” and the state after the clock pulse cannot be A. So a second state, B, is established to represent the occurrence of the first 1 in the sequence. Further, to represent the occurrence of the first 1 in the sequence, a transition is placed from A to B and labeled with a 1. Since this is not the final 1 in the sequence 1101, its output is a 0. This initial portion of the state diagram is given in Figure 6-24 (a).

The next bit of the sequence is a 1. When this 1 occurs in state B, a new state is needed to represent the occurrence of two 1’s in a row on the input—that is, the occurrence of an additional 1 while in state B. So a state C and the associated transition are added, as shown in Figure 6-24 (b). The next bit of the sequence is a 0. When this 0 occurs in state C, a state is needed to represent the occurrence of the two 1’s in a row followed by a 0. So the additional state D with a transition having a 0 input and 0 output is added. Since state D represents the occurrence of 110

as the previous three input bit values on X , the occurrence of a 1 in state D completes the sequence to be recognized, so the transition for the input value 1 from state D has an output value of 1. The resulting partial state diagram, which completely represents the occurrence of the sequence to be recognized, is shown in Figure 6-24(c).

Note in Figure 6-24(c) that, for each state, a transition is specified for only one of the two possible input values. Also, the state that is the destination of the transition from D for input 1 is not yet defined. The remaining transitions must be based on the idea that the recognizer is to identify the sequence 1101, regardless of where it occurs in a longer sequence. Suppose that an initial part of the sequence 1101 is represented by a state in the diagram. Then, the transition from that state for an input value that represents the next input value in the sequence must enter a state such that the 1 output occurs if the remaining bits of the sequence are applied. For example, state C represents the first two bits, 11, of sequence 1101. If the next input value is 0, then the state that is entered, in this case, D, gives a 1 output if the remaining bit of the sequence, 1, is applied.

Next, evaluate where the transition for the 1 input from the D state is to go. Since the transition input is a 1, it could be the first or second bit in the sequence to be recognized. But because the circuit is in state D, it is evident that the prior input was a 0. So this 1 input is the first 1 in the sequence, since it cannot be preceded by a 1. The state that represents the occurrence of a first 1 in the sequence is B, so the transition with input 1 from state D is to state B. This transition is shown in the diagram in Figure 6-24(d). Examining state C, we can trace back through states B and A to see that the occurrence of a 1 input in C is at least the second 1 in the sequence. The state representing the occurrence of two 1's in sequence is C, so the new transition is to state C. Since the combination of two 1's is not the sequence to be recognized, the output for the transition is 0. Repeating this same analysis for missing transitions from states B and A, the final state diagram in Figure 6-24(d) is obtained. The resulting state table is given in two-dimensional form in Table 6-3. ■

One issue that arises in the formulation of any state diagram is whether, in spite of best designer efforts, excess states have been used. This is not the case in the preceding example, since each state represents input history that is essential for

□ **TABLE 6-3**
State Table for State Diagram in Figure 6-21

Present State	Next State		Output Z	
	X = 0	X = 1	X = 0	X = 1
A	A	B	0	0
B	A	C	0	0
C	D	C	0	0
D	A	B	0	1

recognition of the stated sequence. If, however, excess states are present, then it may be desirable to combine states into the fewest needed. This can be done using ad hoc methods and formal state minimization procedures. Due to the complexity of the latter, particularly in the case in which don't-care entries appear in the state table, formal procedures are not covered here. For the interested student, state minimization procedures are found in the references listed at the end of the chapter. The next example illustrates an additional method for avoiding extra states.

EXAMPLE 6-3 Finding a State Diagram for a BCD-to-Excess-3 Decoder

In Chapter 3, a BCD-to-excess-3 decoder was designed. In this example, the function of the circuit is similar except that the inputs, rather than being presented to the circuit simultaneously, are presented serially in successive clock cycles, least significant bit first. In Table 6-4(a), the input sequences and corresponding output sequences are listed with the least significant bit first. For example, during four successive clock cycles, if 1010 is applied to the input, the output will be 0001. In order to produce each output bit in the same clock cycle as the corresponding input bit, the output depends on the present input value as well as the state. The specifications also state that the circuit must be ready to receive a new 4-bit sequence as soon as the prior sequence has completed. The input to this circuit is labeled X and the output is labeled Z . In order to focus on the patterns for past inputs, the rows of Table 6-4(a) are sorted according to the first bit value, the second bit value, and the third bit value of the input sequences. Table 6-4(b) results.

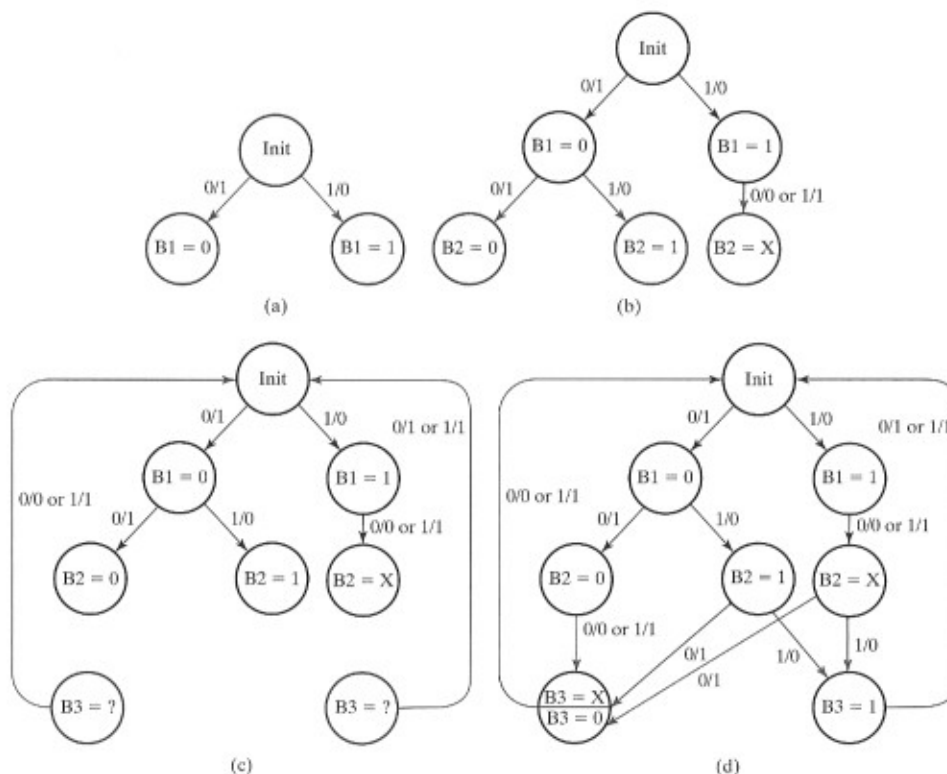
The state diagram begins with an initial state as shown in Figure 6-25(a). Examining the first column of bits in Table 6-4(b) indicates that a 0 produces a 1 output and a 1 produces a 0 output. Next, we ask the question, "Do we need to

□ **TABLE 6-4**
Sequence Tables for Code Converter Example

(a) Sequences in Order of Digits Represented								(b) Sequences in Order of Common Prefixes							
BCD Input				Excess-3 Output				BCD Input				Excess-3 Output			
1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
0	0	0	0	1	1	0	0	0	0	0	0	1	1	0	0
1	0	0	0	0	0	1	0	0	0	0	1	1	1	0	1
0	1	0	0	1	0	1	0	0	0	1	0	1	1	1	0
1	1	0	0	0	1	1	0	0	1	0	0	1	0	1	0
0	0	1	0	1	1	1	0	0	1	1	0	1	0	0	1
1	0	1	0	0	0	0	1	1	0	0	0	0	0	1	0
0	1	1	0	1	0	0	1	1	0	0	1	0	0	1	1
1	1	1	0	0	1	0	1	1	0	1	0	0	0	0	1
0	0	0	1	1	1	0	1	1	1	0	0	0	1	1	0
1	0	0	1	0	0	1	1	1	1	1	0	0	1	0	1

remember the value of the first bit?" In Table 6-4(b), when the first bit is a 0, a 0 in the second bit results in an output of 1 and a 1 in the second bit gives an output of 0. In contrast, if the first bit is a 1, a 0 in the second bit causes an output of 0, and a 1 in the second bit gives output 1. It is clear that the output for the second bit cannot be determined without "remembering" the value of the first bit. Thus, the first input equal to 0 and the first input equal to 1 must give different states as shown in Figure 6-25(a), which also shows the input/output values for the arcs to the new states.

Next, it must be determined whether the inputs following the two new states need to have two states to remember the second bit value. In the first two columns of inputs in Table 6-4(b), sequence 00 produces outputs for the third bit that are 0 for input 0 and 1 for input 1. On the other hand, for sequence 01, the outputs for the third bit are 1 for input 0 and 0 for input 1. Since these are different for the same input values in the third bit, separate states are necessary, as shown in Figure 6-25(b). A similar analysis for input sequences 10 and 11, which examines the outputs for both the third and fourth bits, shows that the value of the second bit has no effect on the output values. Thus, in Figure 6-25(b), there is only a single next state for state B1 = 1.



□ FIGURE 6-25

Construction of a State Diagram for Example 6.3

At this point, six potential new states might result from the three states just added. Note, however, that these states are needed only to define the outputs for the fourth input bit since it is known that the next state thereafter will be *Init* in preparation for applying the next input sequence of four bits. How many states does one need to specify the different possibilities for the output value in the last bit? Looking at the final column, a 1 input always produces a 1 output and a 0 may produce either a 0 or a 1 output. Thus, at most two states are necessary, one that has a 0 output to a 0 and one that has a 1 output to a 0. The output for a 1 input is the same for both states. In Figure 6-25(c), we have added these two states. For the circuit to be ready to receive the next sequence, the next state for these new states is *Init*.

Remaining is the determination of the blue arcs shown in Figure 6-25(d). The arcs from each of the bit B2 states can be defined based on the third bit in the input/output sequences. The next state can be chosen based on the response to input 0 in the fourth bit of the sequence. The B2 state reaches the B3 state on the left with $B3 = 0$ or $B3 = 1$ as indicated by $B3 = X$ on the upper half of the B3 state. The other two B2 states reach this same state with $B3 = 1$ as indicated on the lower half of the state. These same two B2 states reach the B3 state on the right with $B3 = 0$ as indicated by the label on the state. ■

State Assignment

In contrast to the states in the analysis examples, the states in the diagrams constructed have been assigned symbolic names rather than binary codes. It is necessary to replace these symbolic names with binary codes in order to proceed with the design. In general, if there are m states, then the codes must contain n bits, where $2^n \geq m$, and each state must be assigned a unique code. So, for the circuit in Table 6-3 with four states, the codes assigned to the states require two bits.

We begin by assigning a code to the initial reset state. If 1101 occurs as the first four inputs to the circuit after *Reset* = 1, it should be recognized. But if 101, 01, or 1 occurs as the first input sequence, they should not be recognized. The only state that can provide this property is state A. So, with direct resets used on the flip-flops, the code 00 must be assigned to state A. As a basis for encoding the remaining states, extensive work on the assignment of codes to states exists, but it is too complex for our treatment here. These methods have focused primarily on attempting to select codes in such a way that the logic required to implement the flip-flop input equations and output equations is minimized. In our example, we simply assign the state codes in Gray code order, beginning with state A. The Gray code is selected in this case simply because it makes it easier for the next-state and output functions to be placed on a Karnaugh map. The state table with the codes assigned is shown in Table 6-5.

Designing with D Flip-Flops

The remainder of the sequential circuit design procedure will be illustrated by the next example. We wish to design a clocked sequential circuit that operates according to the state table for Example 6-2, the sequence recognizer, shown in Table 6-5.

□ **TABLE 6-5**
Table 6-3 with Names Replaced by Binary Codes

Present State AB	Next State		Output Z	
	X = 0	X = 1	X = 0	X = 1
00	00	01	0	0
01	00	11	0	0
11	10	11	0	0
10	00	01	0	1

This state table, with the binary codes assigned to the states, specifies four states, two input values, and two output values. Two flip-flops are needed to represent the four states. We label the flip-flop outputs with the letters A and B , the input with X , and the output with Z .

Steps 1 through 3 of the design procedure have been completed for this circuit. Beginning step 4, D flip-flops are chosen. To complete step 4, the flip-flop input equations are obtained from the next-state values listed in the table. For step 5, the output equation is obtained from the values of Z in the table. The flip-flop input equations and output equation can be expressed as a sum of minterms of the present-state variables A and B and the input variable X :

$$A(t+1) = D_A(A,B,X) = \Sigma m(3,6,7)$$

$$B(t+1) = D_B(A,B,X) = \Sigma m(1,3,5,7)$$

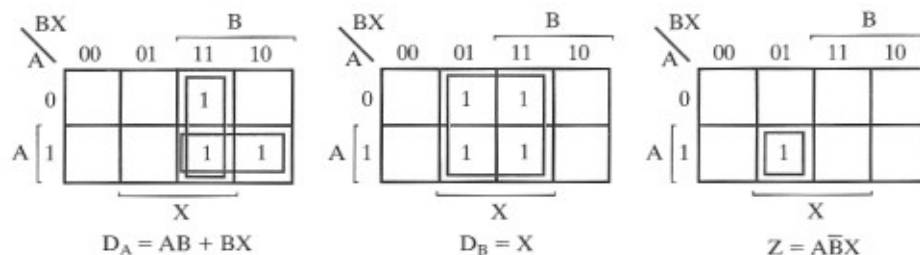
$$Z(A,B,X) = \Sigma m(5)$$

The Boolean functions are simplified by using the maps plotted in Figure 6-26. The simplified functions are

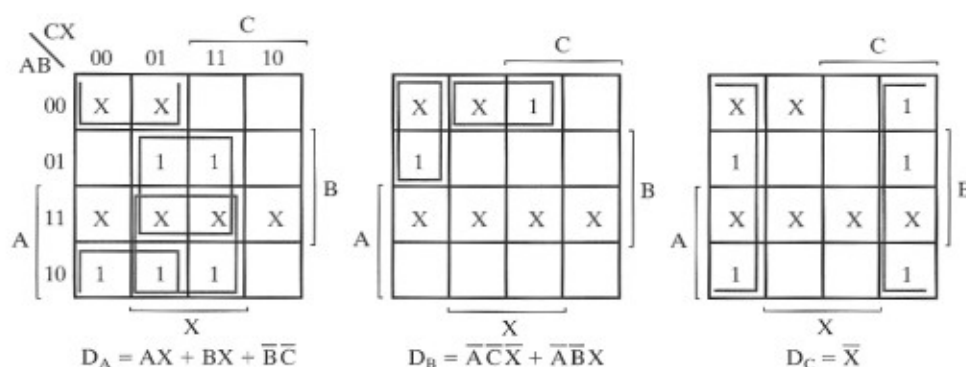
$$D_A = AB + BX$$

$$D_B = X$$

$$Z = A\bar{B}X$$



□ **FIGURE 6-26**
 Maps for Input Equations and Output Z



□ **FIGURE 6-28**
Maps for Optimizing Input Equations

in the table: 000, 110, and 111. When an input of 0 or 1 is included with the unused present-state values, six unused combinations are obtained for the present-state and input columns: 0000, 0001, 1100, 1101, 1110, and 1111. These six combinations are not listed in the state table and hence may be treated as don't-care minterms.

The three input equations for the D flip-flops are derived from the next-state values and are simplified in the maps of Figure 6-28. Each map has six don't-care minterms in the squares corresponding to binary 0, 1, 12, 13, 14, and 15. The optimized equations are

$$D_A = AX + BX + \overline{B}C$$

$$D_B = \overline{A}\overline{C}\overline{X} + \overline{A}BX$$

$$D_C = \overline{X}$$

The logic diagram can be obtained directly from the input equations and will not be drawn here.

It is possible that outside interference or a malfunction will cause the circuit to enter one of the unused states. Thus, it is sometimes desirable to specify, fully or at least partially, the next state values or the output values for the unused states. Depending on the function and application of the circuit, a number of ideas may be applied. First, the outputs for the unused states may be specified so that any actions that result from entry into and transitions between the unused states are not harmful. Second, an additional output may be provided or an unused output code employed which indicates that the circuit has entered an incorrect state. Third, to ensure that a return to normal operation is possible without resetting the entire system, the next-state behavior for the unused states may be specified. Typically, next states are selected such that one of the normally occurring states is reached within a few clock cycles, regardless of the input values. The

decision as to which of the three options to apply, either individually or in combination, is based on the application of the circuit or the policies of a particular design group.

Verification

Sequential circuits can be verified by showing that the circuit produces the original state diagram or state table. In the simplest cases, all possible input combinations are applied with the circuit in each of the states and the state variables and outputs are observed. For small circuits, the actual verification can be performed manually. More generally, simulation is used. In manual simulation, it is straightforward to apply each of the state-input combinations and verify that the output and the next state are correct.

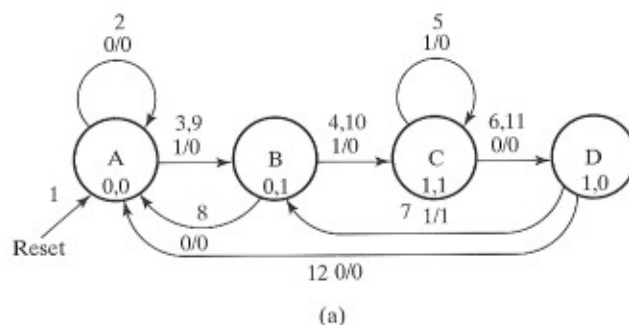
Verification with simulation is less tedious, but typically requires a sequence of input combinations and applied clocks. In order to check out a state-input combination, it is first necessary to apply a sequence of input combinations to place the circuit in the desired state. It is most efficient to find a single sequence to test all the state-input combinations. The state diagram is ideal for generating and optimizing such a sequence. A sequence must be generated to apply each input combination in each state while observing the output and next state that appears after the positive clock edge. The sequence length can be optimized by using the state diagram. The reset signal can be used as an input during this sequence. In particular, it is used at the beginning to reset the circuit to its initial state.

In Example 6-4, both manual and simulation-based verification are illustrated.

EXAMPLE 6-4 Verifying the Sequence Recognizer

The state diagram for the sequence recognizer appears in Figure 6-24(d) and the logic diagram appears in Figure 6-27. There are four states and two input combinations, giving a total of eight state-input combinations to verify. The next state can be observed as the state on the flip-flop outputs after the positive clock edge. For D flip-flops, the next state is the same as the D input just before the clock edge. For other types of flip-flops, the flip-flop inputs just before the clock edge are used to determine the next state of the flip-flop. Initially, beginning with the circuit in an unknown state, we apply a 1 to the Reset input. This input goes to the direct reset input on the two flip-flops in Figure 6-27. Since there is no bubble on these inputs, the 1 value resets both flip-flops to 0, giving state A (0,0). Next, we apply input 0, and manually simulate the circuit in Figure 6-27 to find that the output is 0 and the next state is A (0,0), which agrees with the transition for input 0 while in state A. Next, simulating state A with input 1, next state B (0,1) and output 0 result. For state B, input 0 gives output 0 and next state A (0,0), and input 1 gives output 0 and next state C(1,1). This same process can be continued for each of the two input combinations for states C and D.

For verification by simulation, an input sequence that applies all state-input combination pairs is to be generated accompanied by the output sequence and



Clock Edge:	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Input R:	X	1	0	0	0	0	0	0	0	0	0	0	0	0
Input X:	X	0	0	1	1	1	0	1	0	1	1	0	0	0
State (A,B):	X,X	0,0*	0,0	0,0	0,1	1,1	1,1	1,0	0,1	0,0	0,1	1,1	1,0	0,0
Output Z:	X	0	0	0	0	0	0	1	0	0	0	0	0	0

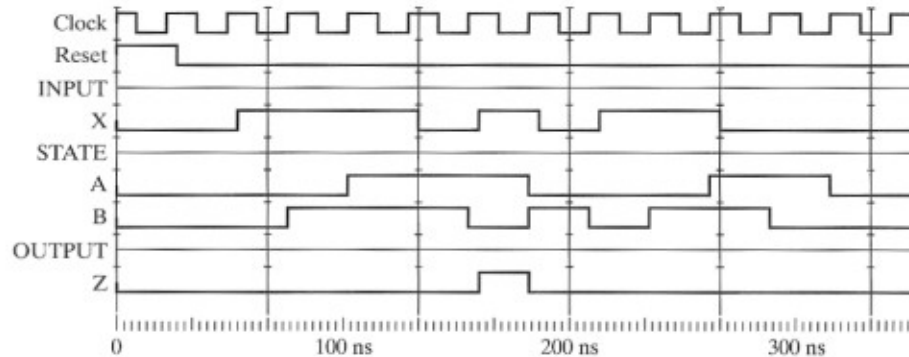
(b)

□ FIGURE 6-29

Test Sequence Generation for Simulation in Example 6.5

state sequence for checking output and next-state values. Optimization requires that the number of clock periods used exceed the number of state–input combination pairs by as few periods as possible (i.e., the repetition of state–input combination pairs should be minimized). This can be interpreted as drawing the shortest path through the state diagram that passes through each state–input combination pair at least once.

In Figure 6-29(a), for convenience, the codes for the states are shown and the path through the diagram is denoted by a sequence of blue integers beginning with 1. These integers correspond to the positive clock edge numbers in Figure 6-29(b), where the verification sequence is to be developed. The values shown for the clock edge numbers are those present just before the positive edge of the clock (i.e., during the setup time interval). Clock edge 0 is at $t = 0$ in the simulation and gives unknown values for all signals. We begin with value 1 applied to Reset (1) to place the circuit in state A. Input value 0 is applied first (2) so that the state remains A, followed by 1 (3) checking the second input combination for state A. Now in state B, we can either move forward to state C or go back to state A. It is not apparent which choice is best, so we arbitrarily apply 1 (4) and go to state C. In state C, 1 is applied (5) so the state remains C. Next, a 0 is applied to check the final input for state C. Now in state D, we have an



□ **FIGURE 6-30**
Simulation for Example 6-5

arbitrary choice to return to state A or to state B. If we return to state B by applying 1 (7), then we can check the transition from B to A for input 0 (8). Then, the only remaining transition to check is state D for input 0. To reach state D from state A, we must apply the sequence 1, 1, 0 (9) (10) (11) and then apply 0 (12) to check the transition from D to A. We have checked eight transitions with a sequence consisting of reset plus 11 inputs. Although this test sequence is of optimum length, optimality is not guaranteed by the procedure used. However, it usually produces an efficient sequence.

In order to simulate the circuit, we enter the schematic in Figure 6-27 using the Xilinx ISE 4.2 Schematic Editor and enter the sequence from Figure 6-29(b) as a waveform using the Xilinx ISE 4.2 HDL Benchner. While entering the waveform, it is important that the input X changes well before the clock edge. This insures that there is time available to display the current output and to permit input changes to propagate to the flip-flop inputs before the setup time begins. This is illustrated by the INPUT waveforms in Figure 6-30 in which X changes shortly after the positive clock edge providing a good portion of the clock period for the change to propagate to the flip-flops. The circuit is simulated with the MTI ModelSim simulator. We can then compare the values just before the positive clock edge on the STATE and OUTPUT waveforms in Figure 6-30 with the values shown on the state diagram for each clock period in Figure 6-29. In this case, the comparison verifies that the circuit operation is correct. ■

6-6 OTHER FLIP-FLOP TYPES

This section introduces *JK* and *T* flip-flops and the representations of their behavior used in analysis and design.



Because of their lesser importance in contemporary design relative to *D* flip-flops, the analysis and design examples illustrating their use are given on the Companion Website for the text.

JK and T Flip-Flops

Four types of flip-flops are characterized in Table 6-7, including the *SR* and *D* from section 6-3 given for reference, and the *JK* and *T* introduced here. With the exception of the *SR* flip-flop which is master-slave, the symbol for a positive-edge-triggered version of each flip-flop type is given. A logic diagram for an implementation of each flip-flop type is either referenced or given. A new concept, the *characteristic table*, defines the logical properties of flip-flop operation in tabular form. Specifically, the table defines the next state as a function of the present state and inputs. $Q(t)$ refers to the present state prior to the application of a clock pulse. $Q(t + 1)$ represents the state one clock period later (i.e., the *next state*). Note that the triggering edge (or pulse) at input *C* is not listed in the characteristic table, but is assumed to occur between time t and $t + 1$. Next to the characteristic table, the *characteristic equation* for each flip-flop type is given. These equations define the next state after the clock pulse for each of the flip-flops as a function of the preset inputs and the present state before the clock pulse. The final column of the table consists of *excitation tables* for each of the flip-flop types. These tables define the input value or values required to obtain each possible next state value after the clock pulse, given the present state value before the clock pulse. Excitation tables can be used to determine the flip-flop input equations from state table information.

Historically, the *JK* flip-flop was a modified version of the master-slave *SR* flip-flop. While the *SR* flip-flop produces undefined outputs and indeterminate behavior for $S = R = 1$, the *JK* flip-flop causes the output to complement its current value. The master-slave version of the *JK* flip-flop has pulse-triggered behavior and, in addition, exhibits a property called “1’s catching.” Once $J = 1$ or $K = 1$ occurs, such that the master changes to the opposite state, the master cannot be changed back to its original state before the clock pulse ends, regardless of the values on *J* and *K*. This worsens the setup time problem that already exists for the pulse-triggered flip-flop. The same solution applies as for the *SR* flip-flop (i.e., making the setup time, t_s , the entire duration of the triggering pulse). To avoid this additional contribution to the length of the clock cycle, we use only edge-triggered *JK* flip-flops built upon an edge-triggered *D* flip-flop.

In Table 6-7, the symbol for a positive-edge-triggered *JK* flip-flop is shown as well as its logic diagram using a positive-edge-triggered *D* flip-flop. The characteristic table given describes the behavior of the *JK* flip-flop. The *J* input behaves like the *S* input to set the flip-flop. The *K* input is similar to the *R* input for resetting the flip-flop. The only difference between the *SR* and *JK* flip-flops is their response to the condition when both inputs are equal to 1. As can be verified from the logic diagram, this condition complements the state of the *JK* flip-flop. When $J = 1$ and $Q = 0$, then $D = 1$, complementing the *JK* flip-flop outputs. When $K = 1$ and $Q = 1$, then $D = 0$, complementing the *JK* flip-flop outputs. This demonstrates that, regardless of the value of Q , the condition $J = 1$ and $K = 1$ causes the outputs of the flip-flop to be complemented in response to a clock pulse. The next state behavior is summarized in the characteristic table column of Table 6-7. The clock input is not explicitly shown, but a clock pulse is assumed to have occurred between the present state and the next state of Q .

□ TABLE 6-7
Flip-Flop Logic, Characteristic Tables and Equations, and Excitation Tables

Type	Symbol	Logic Diagrams	Characteristic Table				Characteristic Equation	Excitation Table				
D		See Figure 6-13	D	Q(t+1)	Operation	$Q(t+1) = D(t)$	Q(t+1)		D	Operation		
			0	0	Reset		0	0	Reset			
			1	1	Set		1	1	Set			
SR		See Figure 6-10	S	R	Q(t+1)	Operation	$Q(t+1) = S(t) + \bar{R}(t) \cdot Q(t)$	Q(t)	Q(t+1)	S	R	Operation
			0	0	Q(t)	No change		0	0	0	X	No change
			0	1	0	Reset		0	1	1	0	Set
			1	0	1	Set		1	0	0	1	Reset
			1	1	?	Undefined		1	1	X	0	No change
JK			J	K	Q(t+1)	Operation	$Q(t+1) = J(t) \cdot \bar{Q}(t) + \bar{K}(t) \cdot Q(t)$	Q(t)	Q(t+1)	J	K	Operation
			0	0	Q(t)	No change		0	0	0	X	No change
			0	1	0	Reset		0	1	1	X	Set
			1	0	1	Set		1	0	X	1	Reset
			1	1	Q̄(t)	Complement		1	1	X	0	No Change
T			T	Q(t+1)	Operation	$Q(t+1) = T(t) \oplus Q(t)$	Q(t+1)		T	Operation		
			0	Q(t)	No change		Q(t)		0	No change		
			1	Q̄(t)	Complement		Q̄(t)		1	Complement		

The T (toggle) flip-flop is equivalent to the JK flip-flop with J and K tied together so that $J = K = T$. With this connection, only the combinations $J = 0, K = 0$ and $J = 1, K = 1$ are applied. If we take the characteristic equation for the JK flip-flop and make this connection, the equation becomes

$$Q(t+1) = T \bar{Q} + \bar{T} Q = T \oplus Q$$

The symbol for the T flip-flop and its logic diagram based on the preceding equation are given in Table 6-7. The characteristic equation for the T flip-flop is that just given, and the characteristic table in Table 6-7 shows that for $T = 0$, the T flip-flop outputs remain unchanged, and for $T = 1$, the outputs are complemented. Since the T flip-flop can only hold its state unchanged or complement its state, there is no way to establish an initial state using only the T input without adding external sampling of the current output in the next state logic outside of the flip-flop. Thus, the T flip-flop is usually initialized to a known state by using a direct set or direct reset.

6-7 HDL REPRESENTATION FOR SEQUENTIAL CIRCUITS—VHDL

In Chapter 4, VHDL was used to describe combinational circuits. Likewise, VHDL can describe storage elements and sequential circuits. In this section, descriptions of a positive-edge-triggered D flip-flop and a sequence recognizer circuit illustrate such uses of VHDL. These descriptions involve new VHDL concepts, the most important of which is the *process*. Thus far, concurrent statements have described combinations of conditions and actions in VHDL. A concurrent statement, however, is limited in the complexity that can be represented. Typically, the sequential circuits to be described are complex enough that description within a concurrent statement is very difficult. A process can be viewed as a replacement for a concurrent statement that permits considerably greater descriptive power. Multiple processes may execute concurrently, and a process may execute concurrently with concurrent statements.

The body of a process typically implements a sequential program. Signal values, which are assigned during the process, however, change only when the process is completed. If the portion of a process executed is

```
B <= A;
C <= B;
```

then, at the completion of the process, B will contain the original contents of A and C will contain the original contents of B . In contrast, after execution of these two statements in a program, C would contain the original contents of A . To achieve program-like behavior, VHDL uses another construct called a *variable*. In contrast to a signal which evaluates after some delay, a variable evaluates immediately. Thus, if B is a variable in the execution of

```
B := A;
C := B;
```

B will instantaneously evaluate to the contents of A and C will evaluate to the new contents of B, so that C finally contains the original contents of A. Variables appear only within processes. Note the use of `:=` instead of `<=` for variable assignment.

EXAMPLE 6-5 VHDL for Positive-Edge-Triggered D Flip-Flop with Reset

Basic process structure is illustrated by an example process describing the architecture of a positive-edge-triggered *D* flip-flop in Figure 6-31. The process begins with the keyword **process**. Optionally, **process** can be preceded by a process name followed by a colon. Following in parentheses are two signals, CLK and RESET. This is the *sensitivity list* for the process. If either CLK or RESET change, then the process is executed. In general, a process is executed whenever a signal or variable in its sensitivity list changes. It is important to note that the sensitivity list is not a parameter list containing all inputs and outputs. For example, D does not appear, since a change in its value cannot initiate a possible change in the value of Q. Following the sensitivity list at the beginning of the process is the keyword **begin**, and at the end of the process the keyword **end** appears. The word **process** following **end** is optional.

Within the body of the process, there are additional VHDL conditional structures that can appear. Notable in the Figure 6-31 example is **if-then-else**. The general structure of an **if-then-else** in VHDL is

```
-- Positive-Edge-Triggered D Flip-Flop with Reset:
-- VHDL Process Description
library ieee;
use ieee.std_logic_1164.all;
entity dff is
    port (CLK, RESET, D : in std_logic;
          Q : out std_logic);
end dff;

architecture pet_pr of dff is
    -- Implements positive edge-triggered bit state storage
    -- with asynchronous reset.

begin
    process (CLK, RESET)
    begin
        if (RESET = '1') then
            Q <= '0';
        elsif (CLK'event and CLK = '1') then
            Q <= D;
        end if;
    end if;
end process;
end;
```

□ FIGURE 6-31

VHDL Process Description of Positive-Edge-Triggered Flip-Flop with Reset

```

if condition then
    sequence of statements
{elsif condition then
    sequence of statements}
else
    sequence of statements
end if;

```

The statements within braces { } can appear from zero to any number of times. The **if-then-else** within a process is similar in effect to the **when else** concurrent assignment statement. Illustrating, we have

```

if A = '1' then
    Q <= X;
elsif B = '0' then
    Q <= Y;
else
    Q <= Z;
end if;

```

If A is 1, then flip-flop Q is loaded with the contents of X. If A is 0 and B is 0, then flip-flop Q is loaded with the contents of Y. Otherwise, Q is loaded with the contents of Z. The end result for the four combination of values on A and B is

```

A = 0, B = 0    Q <= Y
A = 0, B = 1    Q <= Z
A = 1, B = 0    Q <= X
A = 1, B = 1    Q <= X

```

More complex conditional execution of statements can be achieved by nesting if-then-else structures, as in the following code:

```

if A = '1' then
    if C = '0' then
        Q <= W;
    else
        Q <= X;
    end if;
elsif B = '0' then
    Q <= Y;
else
    Q <= Z;
end if;

```

The end result for the eight combinations of values on A, B, and C is

```

A = 0, B = 0, C = 0    Q <= Y
A = 0, B = 0, C = 1    Q <= Y
A = 0, B = 1, C = 0    Q <= Z
A = 0, B = 1, C = 1    Q <= Z
A = 1, B = 0, C = 0    Q <= W
A = 1, B = 0, C = 1    Q <= X

```

```

A = 1, B = 1, C = 0    Q <= W
A = 1, B = 1, C = 1    Q <= X

```

With the information introduced thus far, the positive-edge-triggered *D* flip-flop in Figure 6-31 can now be studied. The sensitivity list for the process includes CLK and RESET, so the process is executed if either CLK or RESET or both change value. If D changes value, the value of Q is not to change for an edge-triggered flip-flop, so D does not appear on the sensitivity list. Based on the **if-then-else**, if RESET is 1, the flip-flop output Q is reset to 0. Otherwise, if the clock value changes, which is represented by appending 'event to CLK, and the new clock value is 1, which is represented by CLK = '1', a positive edge has occurred on CLK. The result of the positive edge occurrence is the loading of the value on D into the flip-flop so that it appears on output Q. Note that, due to the structure of the **if-then-else**, RESET equal to 1 dominates the clocked behavior of the *D* flip-flop causing the output Q to go to 0. Similar simple descriptions can be used to represent other flip-flop types and triggering approaches. ■

EXAMPLE 6-6 VHDL for the Sequence Recognizer

A more complex example in Figures 6-32 and 6-33 represents the sequence recognizer state diagram in Figure 6-24(d). The architecture in this description consists of three distinct processes, which can execute simultaneously and interact via shared signal values. New concepts included are type declarations for defining new types and case statements for handling conditions.

The type declaration permits us to define new types analogous to existing types such as `std_logic`. A type declaration begins with the keyword **type** followed by the name of the new type, the keyword **is**, and, within parentheses, the list of values for signals of the new type. Using the example from Figure 6-31, we have

```
type state_type is (A, B, C, D);
```

The name of the new type is `state_type` and the values in this case are the names of the states in Figure 6-24(d). Once a **type** has been declared, it can be used for declaring signals or variables. From the example in Figure 6-31,

```
signal state, next_state : state_type;
```

indicates that `state` and `next_state` are signals that are of the type `state_type`. Thus, `state` and `next_state` can have values A, B, C, and D.

The basic **if-then-else** (without using the **elsif**) makes a two-way decision based on whether a condition is TRUE or FALSE. In contrast, the **case** statement can make a multiway decision based on which of a number of statements is TRUE. A simplified form for the generic **case** statement is

```

case expression is
  {when choices =>
    sequence of statements;}
end case;

```

```

-- Sequence Recognizer: VHDL Process Description
-- (See Figure 6-24(d) for state diagram)
library ieee;
use ieee.std_logic_1164.all;
entity seq_rec is
    port(CLK, RESET, X: in std_logic;
          Z: out std_logic);
end seq_rec;

architecture process_3 of seq_rec is
    type state_type is (A, B, C, D);
    signal state, next_state : state_type;
begin

    -- Process 1 - state_register: implements positive edge-triggered
    -- state storage with asynchronous reset.
    state_register: process (CLK, RESET)
    begin
        if (RESET = '1') then
            state <= A;
        elsif (CLK'event and CLK = '1') then
            state <= next_state;
        end if;
    end if;
end process;

    -- Process 2 - next_state_function: implements next state as
    -- a function of input X and state.
    next_state_func: process (X, state)
    begin
        case state is
            when A =>
                if X = '1' then
                    next_state <= B;
                else
                    next_state <= A;
                end if;
            when B =>
                if X = '1' then
                    next_state <= C;
                else
                    next_state <= A;
                end if;
        end case;
    end process;
end architecture;

```

□ FIGURE 6-32
VHDL Process Description of a Sequence Recognizer

```

-- Sequence Recognizer: VHDL Process Description (continued)
when C =>
    if X = '1' then
        next_state <= C;
    else
        next_state <= D;
    end if;
when D =>
    if X = '1' then
        next_state <= B;
    else
        next_state <= A;
    end if;
end case;
end process;

-- Process 3 - output_function: implements output as function
-- of input X and state.
output_func: process (X, state)
begin
    case state is
        when A =>
            Z <= '0';
        when B =>
            Z <= '0';
        when C =>
            Z <= '0';
        when D =>
            if X = '1' then
                Z <= '1';
            else
                Z <= '0';
            end if;
        end case;
    end process;
end;

```

□ FIGURE 6-33

VHDL Process Description of a Sequence Recognizer (continued)

The choices must be values that can be taken on by a signal of the type used in the expression. The **case** statement has an effect similar to the **with-select** concurrent assignment statement.

In the example in Figures 6-32 and 6-33, Process 2 uses a **case** statement to define the next state function for the sequence recognizer. The **case** statement makes a multiway decision based on the current state of the circuit, A, B, C, or D. **If-then-else** statements are used for each of the state alternatives to make a binary decision based on whether input X is 1 or 0. Concurrent assignment statements are then used to assign the next state based on the eight possible combinations of state value and input value. For example, consider the state alternative **when B**. If X

equals 1, then the next state will be C; if X equals 0, then the next state will be A. This corresponds to the two transitions out of state B in Figure 6-24(d). For more complex circuits, case statements can also be used for handling the input conditions.

With this brief introduction to the **case** statement, the overall sequencer recognizer can now be studied. Each of the three processes has a distinct function, but the processes interact to provide the overall sequence recognizer. Process 1 describes the storage of the state. Note that the description is like that of the positive-edge-triggered flip-flop. There are two differences, however. The signals involved are of type `state_type` instead of type `std_logic`. Second, the state that results from applying RESET is state A rather than state 0. Also, since we are using state names such as A, B, and C, the number of state variables (i.e., the number of flip-flops) is unspecified and the state codes are unknown. Process 1 is the only one of the three processes that contains storage.

Process 2 describes the next state function, as discussed earlier. The sensitivity list in this case contains signals X and state. In general, for describing combinational logic, all inputs must appear in the sensitivity list, since, whenever an input changes, the process must be executed.

Process 3 describes the output function. The same **case** statement framework as in Process 2 with state as the expression is used. Instead of assigning state names to next state, values 0 and 1 are assigned to Z. If the value assigned is the same for both values 0 and 1 on X, no **if-then-else** is needed, so an **if-then-else** appears only for state D. If there are multiple input variables, more complex **if-then-else** combinations or a **case** statement, as illustrated earlier, can be used to represent the conditioning of the outputs on the inputs. This example is a Mealy state machine, in which the output is a function of the circuit inputs. If it were a Moore state machine, with the output dependent only on the state, input X would not appear on the sensitivity list, and there would be no **if-then-else** structures in the **case** statement. ■

There is a common pitfall present whenever an **if-then-else** or **case** statement is employed. During synthesis, unexpected storage elements in the form of latches or flip-flops appear. For the simple **if-then-else** used in Figure 6-31, using this pitfall gives a specification that synthesizes to a flip-flop. In addition to the two input signals, RESET and CLK, the signal CLK'event is produced by applying the predefined attribute 'event to the CLK signal. CLK'event is TRUE if the value of CLK changes. All possible combinations of values are represented in Table 6-8. Whenever RESET is 0 and the CLK is fixed at 0 or 1 or has a negative edge, no action is specified. In VHDL, it is assumed that, for any combinations of conditions that have unspecified actions in **if-then-else** or **case** statements, the left-hand side of an assignment statement remains unchanged. This is equivalent to $Q \leq Q$, causing storage to occur. Thus, all combinations of conditions must have the resulting action specified when no storage is intended. If this is not a natural situation, an **others** can be used in the **if-then-else** or **case**. If there are binary values used in the **case** statement, just as in Section 4-7, an **others** must also be used to handle combinations including the seven values other than 0 and 1 permitted for `std_logic`.

Together, the three processes used for the sequence recognizer describe the state storage, the next state function, and the output function for a sequential

□ **TABLE 6-8**
Illustration of generation of storage in VHDL

Inputs			Action
RESET = 1	CLK = 1	CLK'event	
FALSE	FALSE	FALSE	Unspecified
FALSE	FALSE	TRUE	Unspecified
FALSE	TRUE	FALSE	Unspecified
FALSE	TRUE	TRUE	$Q \leftarrow D$
TRUE	—	—	$Q \leftarrow '0'$

circuit. Since these are all of the components of a sequential circuit at the state diagram level, the description is complete. The use of three distinct processes is only one methodology for sequential circuit description. Pairs of processes or all three processes can be combined for more elegant descriptions. Nevertheless, the three-process description is the easiest for new users of VHDL and also works well with synthesis tools.

To synthesize the circuit into actual logic, a state assignment is needed, in addition to a technology library. Many synthesis tools will make the state assignment independently or based on a directive from the user. It is also possible for the user to specify explicitly the state assignment. This can be done in VHDL by using an enumeration type. The encoding for the state machine in Figures 6-32 and 6-33 can be specified by adding the following after the **type** *state_type* declaration:

```
attribute enum_encoding: string;
attribute enum_encoding of state_type:
type is "00, 01, 10, 11";
```

This is not a standard VHDL construct, but it is recognized by many synthesis tools. Another option is not to use a type declaration for the states, but to declare the state variables as signals and use the actual codes for the states. In this case, if states appear in the simulation output, they will appear as the encoded state values.

6-8 HDL REPRESENTATION FOR SEQUENTIAL CIRCUITS— VERILOG

In Chapter 4, Verilog was used to describe combinational circuits. Likewise, Verilog can describe storage elements and sequential circuits. In this section, descriptions of a positive edge-triggered D flip-flop and a sequence recognizer circuit illustrate such uses of Verilog. These descriptions will involve new Verilog concepts, the most important of which are the process and the register type for nets.

Thus far, continuous assignment statements have been used to describe combinations of conditions and actions in Verilog. A continuous assignment statement

is limited in what can be described, however. A *process* can be viewed as a replacement for a continuous assignment statement that permits considerably greater descriptive power. Multiple processes may execute concurrently and a process may execute concurrently with continuous assignment statements.

Within a process, procedural assignment statements, which are not continuous assignments, are used. Because of this, the assigned values need to be retained over time. This retention of information can be achieved by using the *register* type rather than the wire type for nets. The keyword for the register type is **reg**. Note that just because a net is of type **reg** does not mean that an actual register is associated with its implementation. There are additional conditions that need to be present to cause an actual register to exist.

There are two basic types of processes, the **initial** process and the **always** process. The **initial** process executes only once, beginning at $t = 0$. The **always** process also executes at $t = 0$, but executes repeatedly thereafter. To prevent rampant, uncontrolled execution, some timing control is needed in the form of delay or event-based waiting. The # operator followed by an integer can be used to specify delay. The @ operator can be viewed as “wait for event.” @ is followed by an expression that describes the event or events the occurrence of which will cause the process to execute.

The body of a process is like a sequential program. The process begins with the keyword **begin** and ends with the keyword **end**. There are procedural assignment statements that make up the body of the process. These assignment statements are classified as blocking or nonblocking. Blocking assignments use = as the assignment operator and nonblocking assignments use <= as the operator. *Blocking assignments* are executed sequentially, much like a program in a procedural language such as C. *Nonblocking assignments* evaluate the right-hand side, but do not make the assignment until all right-hand sides have been evaluated. Blocking assignments can be illustrated by the following process body, in which A, B, and C are of type **reg**:

```
begin
    B = A;
    C = B;
end
```

The first statement transfers the contents of A into B. The second statement then transfers the new contents of B into C. At process completion, C contains the original contents of A.

Suppose that the same process body uses nonblocking assignments:

```
begin
    B <= A;
    C <= B;
end
```

The first statement transfers the original contents of A into B and the second statement transfers the original contents of B into C. At process completion, C contains the original contents of B, not those of A. Effectively, the two statements have executed concurrently instead of in sequence. Nonblocking assignments, except in the cases in which we want registers (of type **reg**) to be evaluated sequentially, will be used.

EXAMPLE 6-7 Verilog for Positive-Edge-Triggered D Flip-Flop with Reset

These new concepts can now be applied to the Verilog description of a positive-edge-triggered *D* flip-flop given in Figure 6-34. The module and its inputs and outputs are declared. *Q* is declared as of type **reg** since it will store information. The process begins with the keyword **always**. Following is **@(posedge CLK or posedge RESET)**. This is the *event control* statement for the process that initiates process execution if an event (i.e., a specified change in a specified signal occurs). For the *D* flip-flop, if either *CLK* or *RESET* changes to 1, then the process is executed. It is important to note that the event control statement is not a parameter list containing all inputs. For example, *D* does not appear, since a change in its value cannot initiate a possible change in the value of *Q*. Following the event control statement at the beginning of the process is the keyword **begin**, and at the end of the process the keyword **end** appears.

Within the body of the process, there are additional Verilog conditional structures that can appear. Notable in the Figure 6-34 example is **if-else**. The general structure of an **if-else** in Verilog is

```

if (condition)
    begin procedural statements end
else if (condition)
    begin procedural statements end
else
    begin procedural statements end

```

If there is a single procedural statement, then the **begin** and **end** are unnecessary:

```

if (A == 1)
    Q <= X;
else if (B == 0)
    Q <= Y;
else
    Q <= Z;

```

Note that a double equals signs is used in conditions. If *A* is 1, then flip-flop *Q* is loaded with the contents of *X*. If *A* is 0 and *B* is 0, then flip-flop *Q* is loaded with the contents of *Y*. Otherwise, *Q* is loaded with the contents of *Z*. The end result for the four combination of values on *A* and *B* is

```

A = 0, B = 0  Q <= Y
A = 0, B = 1  Q <= Z
A = 1, B = 0  Q <= X
A = 1, B = 1  Q <= X

```

The **if-else** within a process is similar in effect to the conditional operator in a continuous assignment statement introduced earlier. The conditional operator can be used within a process, but the **if-else** cannot be used in a continuous assignment statement.

```
// Positive-Edge-Triggered D Flip-Flop with Reset:
// Verilog Process Description

module dff_v(CLK, RESET, D, Q);
    input CLK, RESET, D;
    output Q;
    reg Q;

    always @(posedge CLK or posedge RESET)
    begin
        if (RESET)
            Q <= 0;
        else
            Q <= D;
    end
endmodule
```

□ FIGURE 6-34

Verilog Process Description of Positive-Edge-Triggered Flip-Flop with Reset

More complex conditional execution of statements can be achieved by nesting **if-else** structures. For example, we might have

```
if (A == 1)
    if (C == 0)
        Q <= W;
    else
        Q <= X;
else if (B == 0)
    Q <= Y;
else
    Q <= Z;
```

In this type of structure, an **else** is associated with the closest **if** preceding it that does not already have an **else**. The end result for the eight combinations of values on *A*, *B*, and *C* is

A = 0, B = 0, C = 0	Q <= Y
A = 0, B = 0, C = 1	Q <= Y
A = 0, B = 1, C = 0	Q <= Z
A = 0, B = 1, C = 1	Q <= Z
A = 1, B = 0, C = 0	Q <= W
A = 1, B = 0, C = 1	Q <= X
A = 1, B = 1, C = 0	Q <= W
A = 1, B = 1, C = 1	Q <= X

Returning to the **if-else** in the positive-edge-triggered *D* flip-flop shown in Figure 6-34, assuming that a positive edge has occurred on either *CLK* or *RESET*, if *RESET* is 1, the flip-flop output *Q* is reset to 0. Otherwise, the value on *D* is stored in the flip-flop so that *Q* equals *D*. Due to the structure of the **if-else**, *RESET* equal to 1

dominates the clocked behavior of the *D* flip-flop causing the output *Q* to go to 0. Similar simple descriptions can be used to represent other flip-flop types and triggering approaches. ■

EXAMPLE 6-8 Verilog for the Sequence Recognizer

A more complex example in Figure 6-35 represents the sequence recognizer state diagram in Figure 6-24(d). The architecture in this description consists of three distinct processes that can execute simultaneously and interact via shared signal values. New concepts included are state encoding and case statements for handling conditions.

In Figure 6-35, the module `seq_rec_v` and input and output variables `CLK`, `RESET`, `X`, and `Z` are declared. Next, registers are declared for `state` and `next_state`. Note that since `next_state` need not be stored, it could also be declared as a wire, but, since it is assigned within an **always**, it must be declared as a **reg**. Both registers are two bits, with the most significant bit (MSB) numbered 1 and the least significant bit (LSB) numbered 0.

Next, a name is given to each of the states taken on by `state` and `next_state`, and binary codes are assigned to them. This can be done using a parameter statement or a compiler directive **define**. We will use the parameter statement, since the compiler directive requires a somewhat inconvenient ' before each state throughout the description. From the diagram in Figure 6-24(d), the states are A, B, C, and D. In addition, the parameter statements give the state codes assigned to each of these states. The notation used to define the state codes is `2'b` followed by the binary code. The 2 denotes that there are two bits in the code and the 'b denotes that the base of the code given is binary.

The **if-else** (without using the **else if**) makes a two-way decision based on whether a condition is TRUE or FALSE. In contrast, the **case** statement can make a multiway decision based on which one of a number of statements is TRUE. A simplified form for the generic **case** statement is

```
case expression
  {case expression : statements}
endcase
```

in which the braces { } represent one or more such entries.

The **case expression** must have values that can be taken on by a signal of the type used in **expression**. Typically, there are sequences of multiple statements. In the example in Figure 6-35, the **case** statement for the next state function makes a multiway decision based on the current state of the circuit, A, B, C, or D. For each of the case expressions, conditional statements of various types are used to make a binary decision based on whether input *x* is 1 or 0. Nonblocking assignment statements are then used to assign the next state based on the eight possible combinations of state value and input value. For example, consider the expression B. If *x* equals 1, then the next state will be C; if *x* equals 0, then the next state will be A. This corresponds to the two transitions out of state B in Figure 6-24(d).

With this brief introduction to the **case** statement, the overall sequencer recognizer can now be understood. Each of the three processes has a distinct function, but

```

// Sequence Recognizer: Verilog Process Description
// (See Figure 6-24(d) for state diagram)
module seq_rec_v(CLK, RESET, X, Z);
    input CLK, RESET, X;
    output Z;
    reg [1:0] state, next_state;
    parameter A = 2'b00, B = 2'b01, C = 2'b10, D = 2'b11;
    reg Z;
    // state register: implements positive edge-triggered
    // state storage with asynchronous reset.
    always @(posedge CLK or posedge RESET)
    begin
        if (RESET == 1)
            state <= A;
        else
            state <= next_state;
    end
    // next state function: implements next state as function
    // of X and state
    always @(X or state)
    begin
        case (state)
            A: if (X == 1)
                next_state <= B;
                else
                next_state <= A;
            B: if(X) next_state <= C; else next_state <= A;
            C: if(X) next_state <= C; else next_state <= D;
            D: if(X) next_state <= B; else next_state <= A;
        endcase
    end
    // output function: implements output as function
    // of X and state
    always @(X or state)
    begin
        case (state)
            A: Z <= 0;
            B: Z <= 0;
            C: Z <= 0;
            D: Z <= X ? 1 : 0;
        endcase
    end
endmodule

```

□ **FIGURE 6-35**
Verilog Process Description of a Sequence Recognizer

the processes interact to provide the overall sequence recognizer. The first process describes the state register for storing the sequence recognizer state. Note that the description resembles that of the positive-edge-triggered flip-flop. There are two differences, however. First, there are two bits in the state register. Second, the state that results from applying RESET is state A rather than state 0. The first process is the only one of the three processes that has storage associated with it.

The second process describes the next state function as discussed earlier. The event control statement contains signals X and state. In general, for describing combinational logic, all inputs must appear in the event control statement, since, whenever an input changes, the process must be executed.

The final process describes the output function and uses the same **case** statement framework as in the next state function process. Instead of assigning state names, values 0 and 1 are assigned to Z. If the value assigned is the same for both values 0 and 1 on X, no conditional statement is needed, so a conditional statement appears only for state D. If there are multiple input variables, more complex **if-else** combinations, as illustrated earlier, can be used to represent the conditioning of the outputs on the inputs. This example is a Mealy state machine in which the output is a function of the circuit inputs. If it were a Moore state machine, with the output dependent only on the state, input X would not appear on the event control statement and there would be no conditional structures within the **case** statement. ■

There is a common pitfall present whenever an **if-else** or **case** statement is employed. During synthesis, unexpected storage elements in the form of latches or flip-flops appear. For the very simple **if-else** used in Figure 6-34, this pitfall is employed to give a specification that synthesizes to a flip-flop. In addition to the two input signals, RESET and CLK, events **posedge CLK** and **posedge RESET** are produced, which are TRUE if the value of the respective signal changes from 0 to 1. Selected combinations of values for RESET and the two events are shown in Table 6-9. Whenever RESET has no positive edge, or RESET is 0 and CLK is fixed at 0 or 1 or has a negative edge, no action is specified. In Verilog, the assumption is

□ TABLE 6-9
Illustration of generation of storage in Verilog

Inputs		Action
posedge RESET and RESET = 1	posedge CLK	
FALSE	FALSE	Unspecified
FALSE	TRUE	$Q \leq D$
TRUE	FALSE	$Q \leq 0$
TRUE	TRUE	$Q \leq 0$

that, for any combination of conditions with unspecified actions in **if-else** or **case** statements, the left-hand side of an assignment statement will remain unchanged. This is equivalent to $Q \leq Q$ causing storage to occur. Thus, all combinations of conditions must have the resulting action specified when no storage is intended. To prevent undesirable latches and flip-flops from occurring, for **if-else** structures, care must be taken to include **else** in all cases if storage is not desired. In a **case** statement, a **default** statement which defines what happens for all choices not specified should be added. Within the **default** statement, a specific next state can be specified, which in the example is state A.

Together, the three processes used for the sequence recognizer describe the state storage, the next state function, and the output function for the sequential circuit. Since these are all of the components of a sequential circuit at the state diagram level, the description is complete. The use of three distinct processes is only one methodology for sequential circuit description. For example, the next state and output processes could be easily combined. Nevertheless, the three-process description is the easiest for new users of Verilog and also works well with synthesis tools.

6-9 CHAPTER SUMMARY

Sequential circuits are the foundation upon which most digital design is based. Flip-flops are the basic storage elements for synchronous sequential circuits. Flip-flops are constructed of more fundamental elements called latches. By themselves, latches are transparent and, as a consequence, are very difficult to use in synchronous sequential circuits using a single clock. When latches are combined to form flip-flops, nontransparent storage elements very convenient for use in such circuits are formed. There are two triggering methods used for flip-flops: master-slave and edge triggering. In addition, there are a number of flip-flop types, including *D*, *SR*, *JK*, and *T*.

Sequential circuits are formed using these flip-flops and combinational logic. Sequential circuits can be analyzed to find state tables and state diagrams that represent the behavior of the circuits. Also, analysis can be performed by using logic simulation.

These same state diagrams and state tables can be formulated from verbal specifications of digital circuits. By assigning binary codes to the states and finding flip-flop input equations, sequential circuits can be designed. The design process also includes issues such as finding logic for the circuit outputs, resetting the state at power-up, and controlling the behavior of the circuit when it enters states unused in the original specification. Finally, logic simulation plays an important role in verifying that the circuit designed meets the original specification.

As an alternative to the use of logic diagrams, state diagrams, and state tables, sequential circuits can be defined in VHDL or Verilog descriptions. These descriptions, typically at the behavioral level, provide a powerful, flexible approach to sequential circuit specification for both simulation and automatic circuit synthesis. These representations involve processes that provide added descriptive power beyond the concurrent assignment statements of VHDL and the continuous assignment statement of Verilog. The processes, which permit programlike coding and use

if-then-else and case conditional statements, can also be used to efficiently describe combinational logic.

REFERENCES

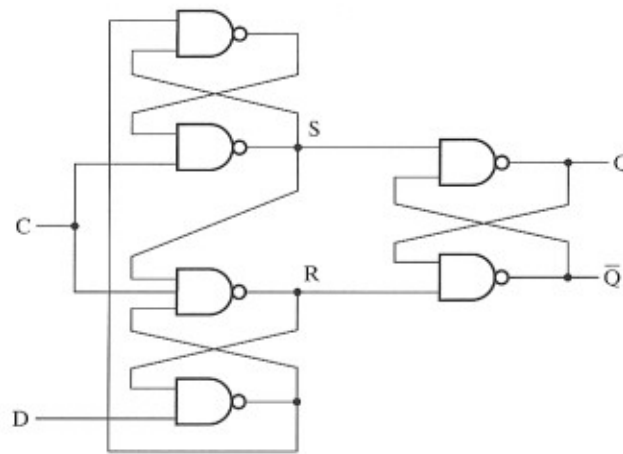
1. MANO, M. M. *Digital Design*, 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 2002.
2. ROTH, C. H. *Fundamentals of Logic Design*, 4th ed. St. Paul: West, 1992.
3. WAKERLY, J. F. *Digital Design: Principles and Practices*, 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2000.
4. *IEEE Standard VHDL Language Reference Manual*. (ANSI/IEEE Std 1076-1993; revision of IEEE Std 1076-1987). New York: The Institute of Electrical and Electronics Engineers, 1994.
5. PELLERIN, D. AND D. TAYLOR. *VHDL Made Easy!* Upper Saddle River, NJ: Prentice Hall PTR, 1997.
6. STEFAN, S. AND L. LINDH. *VHDL for Designers*. London: Prentice Hall Europe, 1997.
7. *IEEE Standard Description Language Based on the Verilog(TM) Hardware Description Language* (IEEE Std 1364-1995). New York: The Institute of Electrical and Electronics Engineers, 1995.
8. PALNITKAR, S. *Verilog HDL: A Guide to Digital Design and Synthesis*. Upper Saddle River, NJ: SunSoft Press (A Prentice Hall Title), 1996.
9. CILETTI, M., *Modeling, Synthesis, and Rapid Prototyping with the Verilog HDL*, Upper Saddle River, NJ: Prentice Hall, 1999.
10. THOMAS, D. E., AND P. R. MOORBY. *The Verilog Hardware Description Language* 4th ed. Boston: Kluwer Academic Publishers, 1998.

PROBLEMS



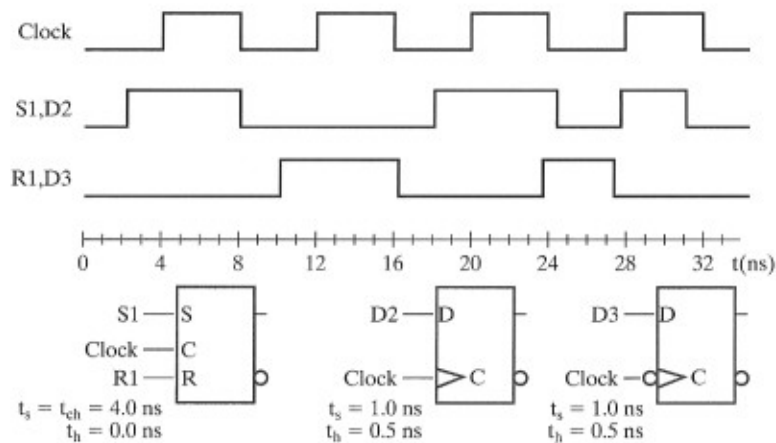
The plus (+) indicates a more advanced problem and the asterisk (*) indicates a solution is available on the Companion Website for the text.

- 6-1. Perform a manual or computer-based logic simulation similar to that given in Figure 6-5 for the \overline{SR} latch shown in Figure 6-6. Construct the input sequence, keeping in mind that changes in state for this type of latch occur in response to 0 rather than 1.
- 6-2. Perform a manual or computer-based logic simulation similar to that given in Figure 6-5 for the SR latch with control input C in Figure 6-7. In particular, examine the behavior of the circuit when S and R are changed while C has the value 1.
- 6-3. A popular alternative design for a positive-edge-triggered D flip-flop is shown in Figure 6-36. Manually or automatically simulate the circuit to determine whether its functional behavior is identical to that of the circuit in Figure 6-13.



□ **FIGURE 6-36**
Circuit for Problem 6-3

- 6-4. A set of waveforms applied to *SR* and *D* flip-flops is shown in Figure 6-37. These waveforms are applied to the flip-flops shown along with the values of their timing parameters.
- Indicate the locations on the waveforms at which there are input combination or timing parameter violations in signal S1 for flip-flop 1.
 - Indicate the locations on the waveforms at which there are input combination or timing parameter violations in signal R1 for flip-flop 1.



□ **FIGURE 6-37**
Waveforms and Flip-flops for Problem 6-4

- (c) List the times at which there are timing parameter violations in signal D2 for flip-flop 2.
- (d) List the times at which there are timing parameter violations in signal D3 for flip-flop 3.

Violations should be indicated even if the state of the flip-flop is such that the violations will not affect the next state.

- 6-5. A sequential circuit with two D flip-flops A and B , two inputs X and Y , and one output Z is specified by the following input equations:

$$D_A = \bar{X}A + XY \quad D_B = \bar{X}A + XB \quad Z = XB$$

- (a) Draw the logic diagram of the circuit.
 - (b) Derive the state table.
 - (c) Derive the state diagram.
- 6-6. *A sequential circuit has three D flip-flops A , B , and C , and one input X . The circuit is described by the following input equations:

$$D_A = (B\bar{C} + \bar{B}C)X + (BC + \bar{B}\bar{C})\bar{X}$$

$$D_B = A$$

$$D_C = B$$

- (a) Derive the state table for the circuit.
 - (b) Draw two state diagrams, one for $X = 0$ and the other for $X = 1$.
- 6-7. A sequential circuit has one flip-flop Q , two inputs X and Y , and one output S . The circuit consists of a D flip-flop with S as its output and logic implementing the function

$$D = X \oplus Y \oplus S$$

with D as the input to the D flip-flop. Derive the state table and state diagram of the sequential circuit.

- 6-8. Starting from state 00 in the state diagram of Figure 6-19(a), determine the state transitions and output sequence that will be generated when an input sequence of 1001101110 is applied.
- 6-9. Draw the state diagram of the sequential circuit specified by the state table in Table 6-10.
- 6-10. *A sequential circuit has two SR flip-flops, one input X , and one output Y . The logic diagram of the circuit is shown in Figure 6-38. Derive the state table and state diagram of the circuit.

- (b) Find the longest path delay in the circuit from an external input to positive clock edge.
 - (c) Find the longest path delay from positive clock edge to output.
 - (d) Find the longest path delay from positive clock edge to positive clock edge.
 - (e) Determine the maximum frequency of operation of the circuit in megahertz (MHz).
- 6-12.** Repeat problem 6-11 assuming that the circuit consists of two copies of the circuit in Figure 6-38 with input X of the second circuit copy driven by input Y of the first circuit copy.
- 6-13.** A sequential circuit is given in Figure 6-17.
- (a) Add the necessary logic and/or connections to the circuit to provide an asynchronous reset to state $A = 0, B = 1$ for signal $\text{Reset} = 1$.
 - (b) Add the necessary logic and/or connections to the circuit to provide a synchronous reset to state $A = 0, B = 0$ for signal $\text{Reset} = 0$.
- 6-14.** *Design a sequential circuit with two D flip-flops A and B and one input X . When $X = 0$, the state of the circuit remains the same. When $X = 1$, the circuit goes through the state transitions from 00 to 10 to 11 to 01, back to 00, and then repeats.
- 6-15.** *A serial two's complementer is to be designed. A binary integer of arbitrary length is presented to the serial two's complementer, least significant bit first, on input X . When a given bit is presented on input X , the corresponding output bit is to appear during the same clock cycle on output Z . To indicate that a sequence is complete and that the circuit is to be initialized to receive another sequence, input Y becomes 1 for one clock cycle. Otherwise, Y is 0.
- (a) Find the state diagram for the serial two's complementer.
 - (b) Find the state table for the serial two's complementer.
- 6-16.** A Universal Serial Bus (USB) communication link requires a circuit that produces the sequence 00000001. You are to design a synchronous sequential circuit that starts producing this sequence for input $E = 1$. Once the sequence starts, it completes. If $E = 1$, during the last output in the sequence, the sequence repeats. Otherwise, if $E = 0$, the output remains constant at 1.
- (a) Draw the Moore state diagram for the circuit.
 - (b) Find the state table and make a state assignment.
 - (c) Design the circuit using D flip-flops and logic gates. A reset should be included to place the circuit in the appropriate initial state at which E is examined to determine if the sequence or constant 1's is to be produced.
- 6-17.** Repeat Problem 6-16 for the sequence 01111110 that is used in a different communication network protocol.
- 6-18.** +The sequence in Problem 6-17 is a flag used in a communication network that represents the beginning of a message. This flag must be unique. As a consequence, at most five 1's in sequence may appear anywhere else in the

message. Since this is unrealistic for normal message content, a trick called zero-insertion is used. The normal message, which can contain strings of 1's longer than 5, enters input X of a sequential zero-insertion circuit. The circuit has two outputs Z and S . When a fifth 1 in sequence appears on X , a 0 is inserted in the stream of outputs appearing on Z and the output $S = 1$ indicating to the circuit supplying the zero-insertion circuit with inputs that it must stall and not apply a new input for one clock cycle. This is necessary since the insertion of 0's in the output sequence causes it to be longer than the input sequence without the stall. Zero-insertion is illustrated by the following example sequences:

Sequence on X without any stalls:	01111100111111100001011110101
Sequence on X with stalls:	01111110011111111100001011110101
Sequence on Z :	0111110001111101100001011110101
Sequence on S :	000000100000001000000000000000

- (a) Find the state diagram for the circuit
- (b) Find the state table for the circuit and make a state assignment.
- (c) Find an implementation of the circuit using D flip-flops and logic gates.

6-19. In many communication and networking systems, the signal transmitted on the communication line uses a non-return-to-zero (NRZ) format. USB uses a specific version referred to as non-return-to-zero inverted (NRZI). A circuit that converts any message sequence of 0's and 1's to a sequence in the NRZI format is to be designed. The mapping for such a circuit is as follows:

- (a) If the message bit is a 0, then the NRZI format message contains an immediate change from 1 to 0 or change from 0 to 1, depending on the current NRZI value.
- (b) If the message bit is a 1, then the NRZI format message remains fixed at 0 or 1, depending on the current NRZI value.

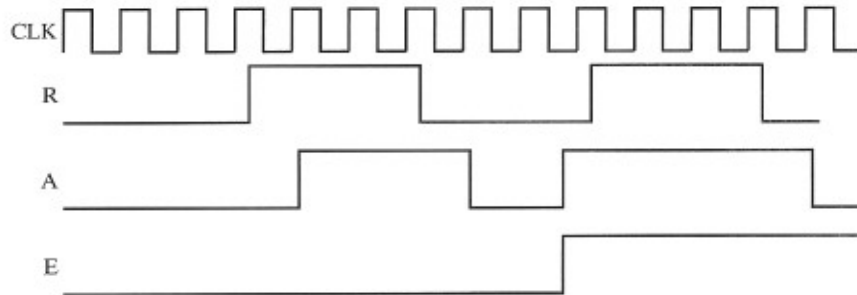
This transformation is illustrated by the following example which assumes that the initial value of the NRZI message is 1:

Message:	10001110011010
NRZI Message:	10100001000101

- (a) Find the Mealy model state diagram for the circuit.
- (b) Find the state table for the circuit and make a state assignment.
- (c) Find an implementation of the circuit using D flip-flops and logic gates.

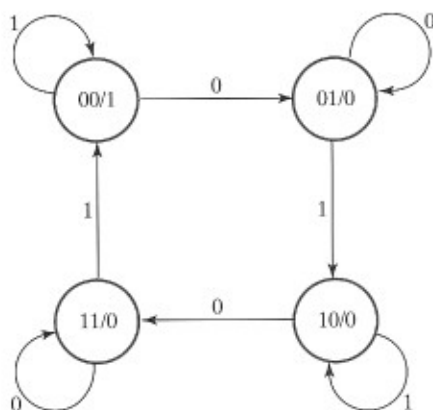
6-20. +Repeat problem 6-19, designing a sequential circuit that transforms an NRZI message into a normal message. The mapping for such a circuit is as follows:

- (a) If a change from 0 to 1 or from 1 to 0 occurs between adjacent bits in the NRZI message, then the message bit is a 0.
- (b) If no change occurs between adjacent bits in the NRZI message, then the message bit is a 1.



□ **FIGURE 6-39**
Signals for Problem 6-21

- 6-21.** A pair of signals Request (R) and Acknowledge (A) is used to coordinate transactions between a CPU and its I/O system. The interaction of these signals is often referred to as a “handshake.” These signals are synchronous with the clock and, for a transaction, are to always have their transitions appear in the order shown in Figure 6-39. A handshake checker is to be designed that will verify the transition order. The checker has inputs, R and A , asynchronous reset signal, RESET, and has output, Error (E). If the transitions in a handshake are in order, $E = 0$. If the transitions are out of order, then E becomes 1 and remains at 1 until the an asynchronous reset signal (RESET = 1) is applied to the CPU.
- Find the state diagram for the handshake checker.
 - Find the state table for the handshake checker.
- 6-22.** A serial leading 1’s detector is to be designed. A binary integer of arbitrary length is presented to the serial leading 1’s detector, most significant bit first, on input X . When a given bit is presented on input X , the corresponding output bit is to appear during the same clock cycle on output Z . As long as the bits applied to X are 0, $Z = 0$. When the first 1 is applied to X , $Z = 1$. For all bit values applied to X after the first 1 is applied, $Z = 0$. To indicate that a sequence is complete and that the circuit is to be initialized to receive another sequence, input Y becomes 1 for one clock cycle. Otherwise, Y is 0.
- Find the state diagram for the serial leading 1’s detector.
 - Find the state table for the serial leading 1’s detector.
- 6-23.** *A sequential circuit has two flip-flops A and B , one input X and one output Y . The state diagram is shown in Figure 6-40. Design the circuit with D flip-flops.
- 6-24.** *A set-dominant master-slave flip-flop has set and reset inputs. It differs from a conventional master-slave SR flip-flop in that, when both S and R are equal to 1, the flip-flop is set.
- Obtain the characteristic table of the set-dominant flip-flop.
 - Find the state diagram for the set-dominant flip-flop.
 - Design the set-dominant flip-flop by using an SR flip-flop and logic gates (including inverters).



□ **FIGURE 6-40**
State Diagram for Problem 6-23

- 6-25.** Find the logic diagram for the circuit having the state table given in Table 6-5. Use *D* flip-flops.
- 6-26.** +The state table for a twisted ring counter is given in Table 6-11. This circuit has no inputs, and its outputs are the uncomplemented outputs of the flip-flops. Since it has no inputs, it simply goes from state to state whenever a clock pulse occurs.
- Design the circuit using *D* flip-flops and assuming that the unspecified next states are don't-care conditions.
 - Add the necessary logic to the circuit to initialize it to state 000 on power-up master reset.
 - In the subsection "Designing with Unused States" of Section 6-5, three techniques for dealing with situations in which a circuit accidentally enters an unused state are discussed. If the circuit you designed in parts (a) and (b) is used in a child's toy, which of the three techniques given would you apply? Justify your decision.

□ **TABLE 6-11**
State Table for Problem 6-26

Present State	Next State
ABC	ABC
000	100
100	110
110	111
111	011
011	001
001	000

- (d) Based on your decision in part (c), redesign the circuit if necessary.
 - (e) Repeat part (c) for the case in which the circuit is used to control engines on a commercial airliner. Justify your decision.
 - (f) Repeat part (d) based on your decision in part (e).
- 6-27. Do a manual verification of the solution (either yours or the one posted on the text website) to Problem 6-24. Consider all transitions where S and R change with the clock equal to 0.
- 6-28. Do an automatic logic simulation-based verification of your design in Problem 6-25. The input sequence used in the simulation should include all transitions in Table 6-6. The simulation output should include the input X and the state variables A , B , and output Z .
- 6-29. *Generate a verification sequence for the circuit described by the state table in Table 6-10. To reduce the length of the simulation sequence, assume that the simulator can handle X inputs and use X 's whenever possible. Assume that a Reset input is available to initialize the state to $A = 0$, $B = 0$ and that all transitions in the state diagram must be exercised.
- 6-30. Design the circuit specified by Table 6-10 and use the sequence from Problem 6-29 (either yours or the one posted on the text web site) to perform an automatic logic simulation-based verification of your design.
- 6-31. *Obtain a timing diagram similar to Figure 6-11 for a positive-edge-triggered JK flip-flop during four clock pulses. Show the timing signals for C , J , K , Y , and Q . Assume that initially the output Q is equal to 1, with $J = 0$ and $K = 1$ for the first pulse. Then, for successive pulses, J goes to 1, followed by K going to 0 and then J going back to 0. Assume that each input changes near the negative edge of the pulse.



All files referred to in the remaining problems are available in ASCII form for simulation and editing on the Companion Website for the text. A VHDL or Verilog compiler/simulator is necessary for the problems or portions of problems requesting simulation. Descriptions can still be written, however, for many of the problems without using compilation or simulation.

- 6-32. *Write a VHDL description for the multiplexer in Figure 4-14 by using a process containing a case statement rather than the continuous assignment statements as shown in Section 4-7.
- 6-33. Repeat Problem 6-32 by using a VHDL process containing if-then-else statements.
- 6-34. +Write a VHDL description for the sequential circuit with the state diagram given by Figure 6-25(d). Include an asynchronous RESET signal to initialize the circuit to state `Init`. Compile your description, apply an input sequence to pass through every arc of the state diagram at least once, and verify the correctness of the state and output sequence by comparing them to the state diagram.

- 6-35. Write a VHDL description for the circuit specified in Problem 6-15.
- 6-36. Write a VHDL description for the circuit specified in Problem 6-19.
- 6-37. *Write a VHDL description for a JK negative-edge-triggered flip-flop with clock CLK. Compile and simulate your description. Apply a sequence that causes all eight combinations of inputs J and K and stored value Q to be applied in some clock cycle.
- 6-38. Write a Verilog description for the multiplexer in Figure 4-14 by using a process containing a case statement rather than the continuous assignment statements as shown in Section 4-8.
- 6-39. *Repeat Problem 6-38 by using a Verilog process containing if-else statements.
- 6-40. +Write a Verilog description for the sequential circuit given by the state diagram in Figure 6-25(d). Include an asynchronous RESET signal to initialize the circuit to state `Init`. Compile your description, apply an input sequence to pass through every arc of the state diagram at least once, and verify the correctness of the state and output sequence by comparing them to the state diagram.
- 6-41. Write a Verilog description for the circuit specified in Problem 6-15.
- 6-42. Write a Verilog description for the circuit specified in Problem 6-19.
- 6-43. *Write a Verilog description for a JK negative-edge-triggered flip-flop with clock CLK. Compile and simulate your description. Apply a sequence that causes all eight combinations of inputs J and K and stored value Q to be applied in some clock cycle.

REGISTERS AND REGISTER TRANSFERS

In Chapters 4 and 5, we studied combinational functional blocks. In Chapter 6, we examined sequential circuits. In this chapter, we bring the two ideas together and present sequential functional blocks, generally referred to as registers and counters. In Chapter 6, the circuits that were analyzed or designed did not have any particular structure, and the number of flip-flops was quite small. In contrast, the circuits we consider here have more structure, with multiple stages or cells that are identical or close to identical. Also, because of this structure, it is easy to add more stages to produce circuits with many more flip-flops than the circuits in Chapter 6. Registers are particularly useful for storing information during the processing of data and counters assist in sequencing the processing.

In a digital system, a datapath and a control unit are frequently present at the top levels of the design hierarchy. A *datapath* consists of processing logic and a collection of registers that performs data processing. A *control unit* is made up of logic that determines the sequence of data-processing operations performed by the datapath. Register transfer notation describes elementary data-processing actions referred to as *microoperations*. Register transfers move information between registers, between registers and memory, and through processing logic. Dedicated transfer hardware using multiplexers and shared transfer hardware called *buses* implement these movements of data.

In the generic computer at the beginning of Chapter 1, registers are used extensively for temporary storage of data in areas aside from memory. Registers of this kind are often large, with at least 32 bits. Special registers called shift registers are used less frequently, appearing primarily in the input-output parts of the system. Counters are used in the various parts of the computer to control or keep track of the sequence of activities. Overall, sequential functional blocks are used widely in the generic computer. In particular, the CPU and FPU parts of the processor each contain large numbers of registers that are involved in register transfers and execution of microoperations. It is in the CPU and the FPU that data transfers, additions, subtractions, and other microoperations take place. Finally, the connections shown between various electronic parts of the computer are buses, which we discuss for the first time in this chapter.

7-1 REGISTERS AND LOAD ENABLE

A register includes a set of flip-flops. Since each flip-flop is capable of storing one bit of information, an n -bit register, composed of n flip-flops, is capable of storing n bits of binary information. By the broadest definition, a *register* consists of a set of flip-flops, together with gates that implement their state transitions. This broad definition includes the various sequential circuits considered in Chapter 6. More commonly, the term *register* is applied to a set of flip-flops, possibly with added combinational gates, that perform data-processing tasks. The flip-flops hold data, and the gates determine the new or transformed data to be transferred into the flip-flops.

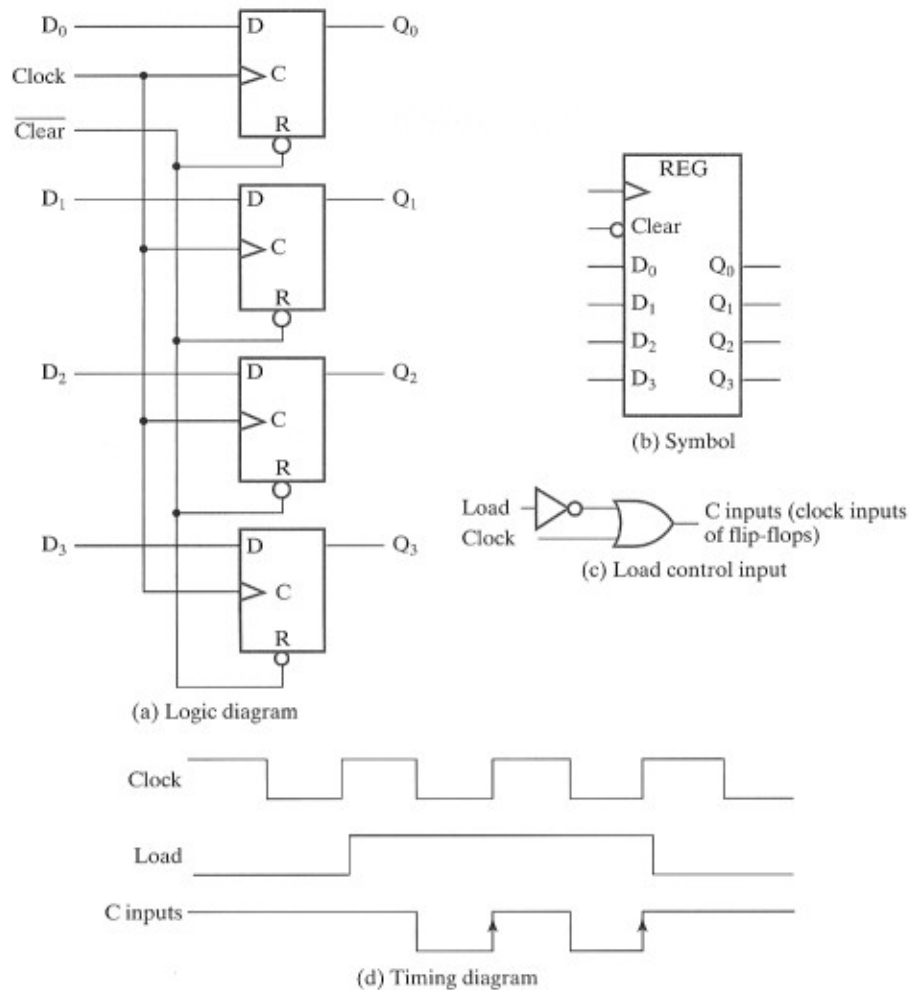
A *counter* is a register that goes through a predetermined sequence of states upon the application of clock pulses. The gates in the counter are connected in a way that produces the prescribed sequence of binary states. Although counters are a special type of registers, it is common to differentiate them from registers.

Registers and counters are sequential functional blocks that are used extensively in the design of digital systems in general and in digital computers in particular. Registers are useful for storing and manipulating information; counters are employed in circuits that sequence and control operations in a digital system.

The simplest register is a register that consists of only flip-flops without external gates. Figure 7-1(a) shows such a register constructed from four D -type flip-flops. The common *Clock* input triggers all flip-flops on the rising edge of each pulse, and the binary information available at the four D inputs is transferred into the 4-bit register. The four Q outputs can be sampled to obtain the binary information stored in the register. The *Clear* input goes to the \bar{R} inputs of all four flip-flops and is used to clear the register to all 0's prior to its clocked operation. This input is labeled *Clear* rather than *Clear*, since a 0 must be applied to reset the flip-flops asynchronously. Activation of the asynchronous \bar{R} inputs to flip-flops during normal clocked operation can lead to circuit designs that are highly delay dependent and that can, therefore, easily malfunction. Thus, we maintain *Clear* at logic 1 during normal clocked operation, allowing it to be logic 0 only when a system reset is desired. We note that the ability to clear a register to all 0's is optional; whether a clear operation is provided depends upon the use of the register in the system.

The transfer of new information into a register is referred to as *loading* the register. If all the bits of the register are loaded simultaneously with a common clock pulse, we say that the loading is done in parallel. A positive clock transition applied to the *Clock* input of the register of Figure 7-1(a) loads all four D inputs into the flip-flops in parallel.

Figure 7-1(b) shows a symbol for the register in Figure 7-1(a). This symbol permits the use of the register in a design hierarchy. The symbol has all inputs to the logic circuit on its left and all outputs from the circuit on the right. The inputs include the clock input with the dynamic indicator to represent positive-edge triggering of the flip-flops. We note that the name *Clear* appears inside the symbol, with a bubble in the signal line on the outside of the symbol. This notation indicates that application of a logic 0 to the signal line activates the clear operation on



□ **FIGURE 7-1**
4-Bit Register

the flip-flops in the register. If the signal line were labeled outside the symbol, the label would be *Clear*.

Register with Parallel Load

Most digital systems have a master clock generator that supplies a continuous train of clock pulses. The pulses are applied to all flip-flops and registers in the system. In effect, the master clock acts like a heart that supplies a constant beat to all parts of the system. For the design in Figure 7-1(a), the clock must be prevented from reaching the clock input to the circuit if the contents of the register are to be left unchanged. Thus, a separate control signal is used to control the

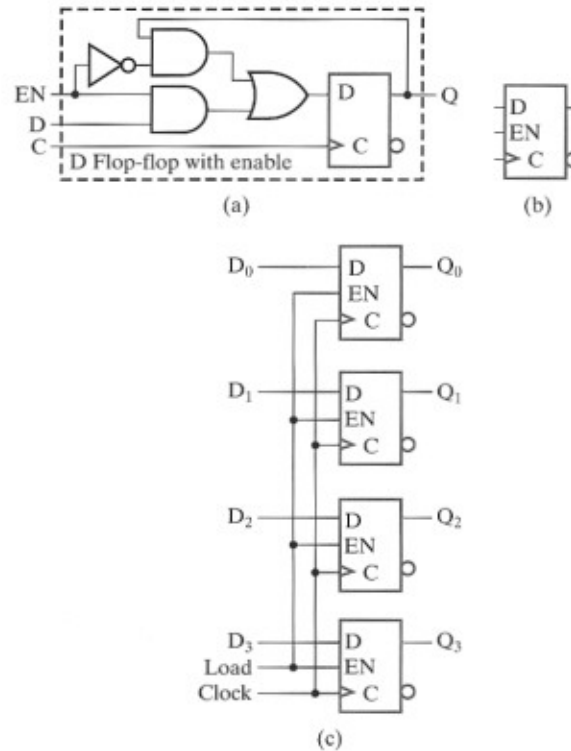
clock cycles during which clock pulses are to have an effect on the register. The clock pulses are prevented from reaching the register when its content is not to be changed. This approach can be implemented with a load control input *Load* combined with the clock, as shown in Figure 7-1(c). The output of the OR gate is applied to the *C* inputs of the register flip-flops. The equation for the logic shown is

$$C \text{ inputs} = \overline{Load} + Clock$$

When the *Load* signal is 1, $C \text{ inputs} = Clock$, so the register is clocked normally, and new information is transferred into the register on the positive transitions of the clock. When the *Load* signal is 0, $C \text{ inputs} = 1$. With this constant input applied, there are no positive transitions on *C inputs*, so the contents of the register remain unchanged. The effect of the *Load* signal on the signal *C inputs* is shown in Figure 7-1(d). Note that the clock pulses that appear on *C inputs* are pulses to 0, which end with the positive edge that triggers the flip-flops. These pulses and edges appear when *Load* is 1 and are replaced by a constant 1 when *Load* is 0. In order for this circuit to work correctly, *Load* must be constant at the correct value, either 0 or 1, throughout the interval when *Clock* is 0. One situation in which this occurs is if *Load* comes from a flip-flop that is triggered on a positive edge of *Clock*, a normal circumstance if all flip-flops in the system are positive-edge triggered. Since the clock is turned on and off at the register *C* inputs by the use of a logic gate, the technique is referred to as *clock gating*.

Inserting gates in the clock pulse path produces different propagation delays between *Clock* and the inputs of flip-flops with and without clock gating. If the clock signals arrive at different flip-flops or registers at different times, *clock skew* is said to exist. But to have a truly synchronous system, we must ensure that all clock pulses arrive simultaneously throughout the system so that all flip-flops trigger at the same time. For this reason, in routine designs, control of the operation of the register without using clock gating is advisable. Otherwise, delays must be controlled to drive the clock skew as close to zero as possible. This is applicable in aggressive low power or high speed designs.

A 4-bit register with a control input *Load* that is directed through gates into the *D* inputs of the flip-flops, instead of through the *C* inputs, is shown in Figure 7-2(c). This register is based on a bit cell shown in Figure 7-2(a) consisting of a 2-to-1 multiplexer and a *D* flip-flop. The signal *EN* selects between the data bit *D* entering the cell and the value *Q* at the output of the cell. For $EN = 1$, *D* is selected and the cell is loaded. For $EN = 0$, *Q* is selected and the output is loaded back into the flip-flop, preserving its current state. The feedback connection from output to input of the flip-flop is necessary because the *D* flip-flop, unlike other flip-flop types does not have a “no change” input condition: With each clock pulse, the *D* input determines the next state of the output. To leave the output unchanged, it is necessary to make the *D* input equal to the present value of the output. The logic in Figure 7-2(a) can be viewed as a new type of *D* flip-flop, a *D flip-flop with enable*, having the symbol shown in Figure 7-2(b).



□ **FIGURE 7-2**
4-Bit Register with Parallel Load

The register is implemented by placing four D flip-flops with enables in parallel and connecting the *Load* input to the *EN* inputs. When *Load* is 1, the data on the four inputs is transferred into the register with the next positive clock edge. When *Load* is 0, the current value remains in the register at the next positive clock edge. Note that the clock pulses are applied continuously to the *C* inputs. *Load* determines whether the next pulse accepts new information or leaves the information in the register intact. The transfer of information from inputs to register is done simultaneously for all four bits during a single positive pulse transition. This method of transfer is traditionally preferred over clock gating, since it avoids clock skew and the potential for malfunctions of the circuit.

7-2 REGISTER TRANSFERS

A digital system is a sequential circuit made up of interconnected flip-flops and gates. In Chapter 6, we learned that sequential circuits can be specified by means of state tables. To specify a large digital system with state tables is very difficult, if not impossible, because the number of states is prohibitively large. To overcome this difficulty, digital systems are designed using a modular, hierarchical approach. The system is partitioned into subsystems or modules, each of which performs some

functional task. The modules are constructed hierarchically from functional blocks such as registers, counters, decoders, multiplexers, buses, arithmetic elements, flip-flops, and primitive gates. The various subsystems communicate with data and control signals to form a digital system.

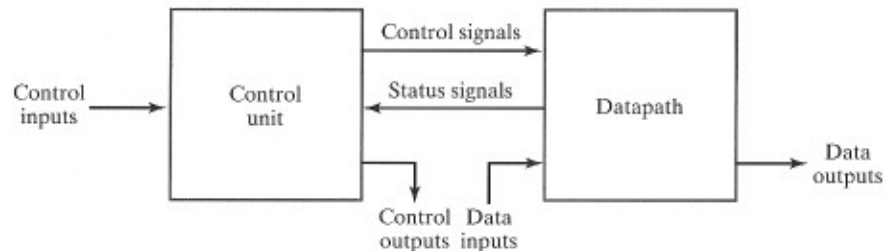
In most digital system designs, we partition the system into two types of modules: a *datapath*, which performs data-processing operations, and a *control unit*, which determines the sequence of those operations. Figure 7-3 shows the general relationship between a datapath and a control unit. *Control signals* are binary signals that activate the various data-processing operations. To activate a sequence of such operations, the control unit sends the proper sequence of control signals to the datapath. The control unit, in turn, receives status bits from the datapath. These status bits describe aspects of the state of the datapath. The status bits are used by the control unit in defining the specific sequence of the operations to be performed. Note that the datapath and control unit may also interact with other parts of a digital system, such as memory and input-output logic, through the paths labeled data inputs, data outputs, control inputs, and control outputs.

Datapaths are defined by their registers and the operations performed on binary data stored in the registers. Examples of register operations are load, clear, shift, and count. The registers are assumed to be basic components of the digital system. The movement of the data stored in registers and the processing performed on the data are referred to as *register transfer operations*. The register transfer operations of digital systems are specified by the following three basic components:

1. the set of registers in the system,
2. the operations that are performed on the data stored in the registers, and
3. the control that supervises the sequence of operations in the system.

A register has the capability to perform one or more *elementary operations* such as load, count, add, subtract, and shift. For example, a right-shift register is a register that can shift data to the right. A counter is a register that increments a number by one. A single flip-flop is a 1-bit register that can be set or cleared. In fact, by this definition, the flip-flops and closely associated gates of any sequential circuit can be called registers.

An elementary operation performed on data stored in registers is called a *microoperation*. Examples of microoperations are loading the contents of one



□ **FIGURE 7-3**
Interaction between Datapath and Control Unit

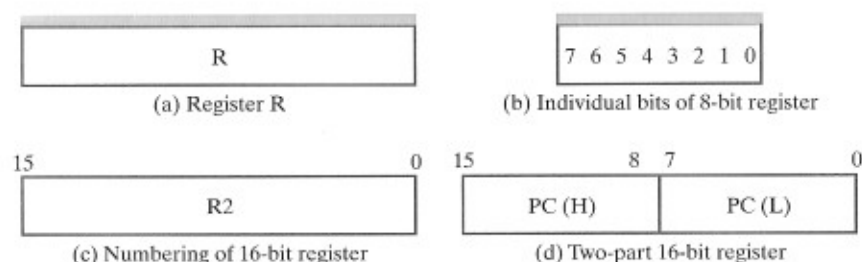
register into another, adding the contents of two registers, and incrementing the contents of a register. A microoperation is usually, but not always, performed in parallel on a vector of bits during one clock cycle. The result of the microoperation may replace the previous binary data in the register. Alternatively, the result may be transferred to another register, leaving the previous data unchanged. The sequential functional blocks introduced in this chapter are registers that implement one or more microoperations.

The control unit provides signals that sequence the microoperations in a prescribed manner. The results of a current microoperation may determine both the sequence of control signals and the sequence of future microoperations to be executed. Note that the term “microoperation,” as used here, does not refer to any particular way of producing the control signals: specifically, it does not imply that the control signals are generated by a control unit based on a technique called microprogramming.

This chapter introduces registers, their implementations and register transfers using a simple register transfer language (RTL) to represent registers and specify the operations on their contents. The register transfer language uses a set of expressions and statements that resemble statements used in HDLs and programming languages. This notation can concisely specify part or all of a complex digital system such as a computer. The specification then serves as a basis for a more detailed design of the system.

7-3 REGISTER TRANSFER OPERATIONS

We denote the registers in a digital system by uppercase letters (sometimes followed by numerals) that indicate the function of the register. For example, a register that holds an address for the memory unit is usually called an address register and can be designated by the name *AR*. Other designations for registers are *PC* for program counter, *IR* for instruction register, and *R2* for register 2. The individual flip-flops in an *n*-bit register are typically numbered in sequence from 0 to *n*-1, starting with 0 in the least significant (often the rightmost) position and increasing toward the most significant position. Since the 0 bit is on the right, this order can be referred to as *little-endian*, in the same manner as for bytes in Chapter 1. The reverse order, with bit 0 on the left, is referred to as *big-endian*. Figure 7-4 shows representations of registers in block diagram form. The most common way to represent a register is by a rectangular box with the name of the register inside, as in part (a) of the figure. The individual bits can be identified as in part (b). The numbering of bits represented by just the leftmost and rightmost values at the top of a register box is illustrated by a 16-bit register *R2* in part (c). A 16-bit program counter, *PC*, is partitioned into two sections in part (d) of the figure. In this case, bits 0 through 7 are assigned the symbol *L* (for low-order byte), and bits 8 through 15 are assigned the symbol *H* (for high-order byte). The label *PC(L)*, which may also be written *PC(7:0)*, refers to the low-order byte of the register, and *PC(H)* or *PC(15:8)* refers to the high-order byte.



□ **FIGURE 7-4**
Block Diagrams of Registers

Data transfer from one register to another is designated in symbolic form by means of the replacement operator (\leftarrow). Thus, the statement

$$R2 \leftarrow R1$$

denotes a transfer of the contents of register $R1$ into register $R2$. In other words, the statement designates the copying of the contents of $R1$ into $R2$. The register $R1$ is referred to as the *source* of the transfer and the register $R2$ as the *destination*. By definition, the contents of the source register do not change as a result of the transfer; only the contents of the destination register, $R2$, change.

A statement that specifies a register transfer implies that datapath circuits are available from the outputs of the source register to the inputs of the destination register and that the destination register has a parallel load capability. Normally, we want a given transfer to occur not for every clock pulse, but only for specific values of the control signals. This can be specified by a *conditional statement*, symbolized by the *if-then* form

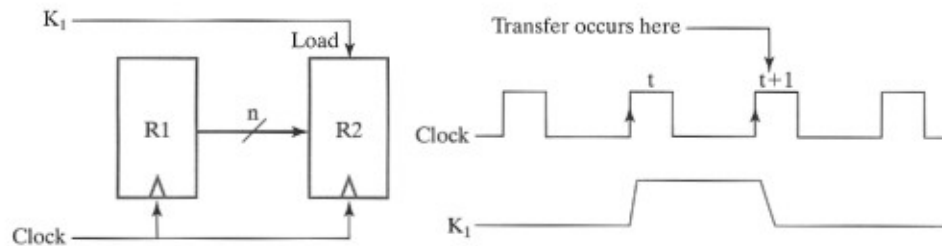
$$\text{if}(K_1 = 1)\text{then}(R2 \leftarrow R1)$$

where K_1 is a control signal generated in the control unit. In fact, K_1 can be any Boolean function that evaluates to 0 or 1. A more concise way of writing the if-then form is

$$K_1: R2 \leftarrow R1$$

This control condition, terminated with a colon, symbolizes the requirement that the transfer operation be executed by the hardware only if $K_1 = 1$.

Every statement written in register transfer notation presupposes a hardware construct for implementing the transfer. Figure 7-5 shows a block diagram that depicts the transfer from $R1$ to $R2$. The n outputs of register $R1$ are connected to the n inputs of register $R2$. The letter n is used to indicate the number of bits in the register transfer path from $R1$ to $R2$. When the width of the path is known, n is replaced by an actual number. Register $R2$ has a load control input that is activated by the control signal K_1 . It is assumed that the signal is synchronized with the same clock as the one applied to the register. The flip-flops are assumed to be positive-edge triggered by this clock. As shown in the timing diagram, K_1 is set to 1 on



□ **FIGURE 7-5**
Transfer from R1 to R2 when $K_1 = 1$

the rising edge of a clock pulse at time t . The next positive transition of the clock at time $t + 1$ finds $K_1 = 1$, and the inputs of R2 are loaded into the register in parallel. In this case, K_1 returns to 0 on the positive clock transition at time $t + 1$, so that only a single transfer from R1 to R2 occurs.

Note that the clock is not included as a variable in the register transfer statements. It is assumed that all transfers occur in response to a clock transition. Even though the control condition K_1 becomes active at time t , the actual transfer does not occur until the register is triggered by the next positive transition of the clock, at time $t + 1$.

The basic symbols we use in register transfer notation are listed in Table 7-1. Registers are denoted by an uppercase letter, possibly followed by one or more uppercase letters and numerals. Parentheses are used to denote a part of a register by specifying the range of bits in the register or by giving a symbolic name to a portion of the register. The left-pointing arrow denotes a transfer of data and the direction of transfer. A comma is used to separate two or more register transfers that are executed at the same time. For example, the statement

$$K_3: R2 \leftarrow R1, R1 \leftarrow R2$$

denotes an operation that exchanges the contents of two registers simultaneously for a positive clock edge at which $K_3 = 1$. Such an exchange is possible with registers made of flip-flops, but presents a difficult timing problem with registers made

□ **TABLE 7-1**
Basic Symbols for Register Transfers

Symbol	Description	Examples
Letters (and numerals)	Denotes a register	$AR, R2, DR, IR$
Parentheses	Denotes a part of a register	$R2(1), R2(7:0), AR(L)$
Arrow	Denotes transfer of data	$R1 \leftarrow R2$
Comma	Separates simultaneous transfers	$R1 \leftarrow R2, R2 \leftarrow R1$
Square brackets	Specifies an address for memory	$DR \leftarrow M[AR]$


TABLE 7-2
Textbook RTL, VHDL, and Verilog Symbols for Register Transfers

Operation	Text RTL	VHDL	Verilog
Combinational Assignment	=	<= (concurrent)	assign = (nonblocking)
Register Transfer	←	<= (concurrent)	<= (nonblocking)
Addition	+	+	+
Subtraction	−	−	−
Bitwise AND	^	and	&
Bitwise OR	∨	or	
Bitwise XOR	⊕	xor	^
Bitwise NOT	~	not	~
Shift left (logical)	sl	sll	<<
Shift right (logical)	sr	srl	>>
Vectors/Registers	A(3:0)	A(3 downto 0)	A[3:0]
Concatenation		&	{ , }

of latches. Square brackets are used in conjunction with a memory transfer. The letter *M* designates a memory word, and the register enclosed inside the square brackets provides the address of the word in memory. This is explained in more detail in Chapter 10.

7-4 A NOTE FOR VHDL AND VERILOG USERS ONLY

Although there are some similarities, the register transfer language used here differs from both VHDL and Verilog. In particular, there is different notation used in each of the three languages. Table 7-2 compares the notation for many identical or similar register transfer operations in the three languages. As you study this chapter and others to follow, this table will assist you in relating descriptions in the text RTL to the corresponding descriptions in VHDL or Verilog.

7-5 MICROOPERATIONS

A microoperation is an elementary operation performed on data stored in registers or in memory. The microoperations most often encountered in digital systems are of four types:

1. *Transfer* microoperations, which transfer binary data from one register to another.
2. *Arithmetic* microoperations, which perform arithmetic on data in registers.
3. *Logic* microoperations, which perform bit manipulation on data in registers.
4. *Shift* microoperations, which shift data in registers.

A given microoperation may be of more than one type. For example, a 1's complement operation is both an arithmetic microoperation and a logic microoperation.

Transfer microoperations were introduced in the previous section. This type of microoperation does not change the binary data bits as they move from the source register to the destination register. The other three types of microoperations can produce new binary data and, hence, new information. In digital systems, basic sets of operations are used to form sequences that implement more complicated operations. In this section, we define a basic set of microoperations, symbolic notation for these microoperations, and descriptions of the digital hardware that implements them.

Arithmetic Microoperations

We define the basic arithmetic microoperations as add, subtract, increment, decrement, and complement. The statement

$$R0 \leftarrow R1 + R2$$

specifies an add operation. It states that the contents of register $R2$ are to be added to the contents of register $R1$ and the sum transferred to register $R0$. To implement this statement with hardware, we need three registers and a combinational component that performs the addition, such as a parallel adder. The other basic arithmetic operations are listed in Table 7-3. Subtraction is most often implemented through complementation and addition. Instead of using the minus operator, we can specify 2's complement subtraction by the statement

$$R0 \leftarrow R1 + \overline{R2} + 1$$

where $\overline{R2}$ specifies the 1's complement of $R2$. Adding 1 to $\overline{R2}$ gives the 2's complement of $R2$. Finally, adding the 2's complement of $R2$ to the contents of $R1$ is equivalent to $R1 - R2$.

The increment and decrement microoperations are symbolized by a plus-one and minus-one operation, respectively. These operations are implemented by using

□ **TABLE 7-3**
Arithmetic Microoperations

Symbolic designation	Description
$R0 \leftarrow R1 + R2$	Contents of $R1$ plus $R2$ transferred to $R0$
$R2 \leftarrow \overline{R2}$	Complement of the contents of $R2$ (1's complement)
$R2 \leftarrow \overline{R2} + 1$	2's complement of the contents of $R2$
$R0 \leftarrow R1 + \overline{R2} + 1$	$R1$ plus 2's complement of $R2$ transferred to $R0$ (subtraction)
$R1 \leftarrow R1 + 1$	Increment the contents of $R1$ (count up)
$R1 \leftarrow R1 - 1$	Decrement the contents of $R1$ (count down)

a special combinational circuit, an adder-subtractor, or a binary up-down counter with parallel load.

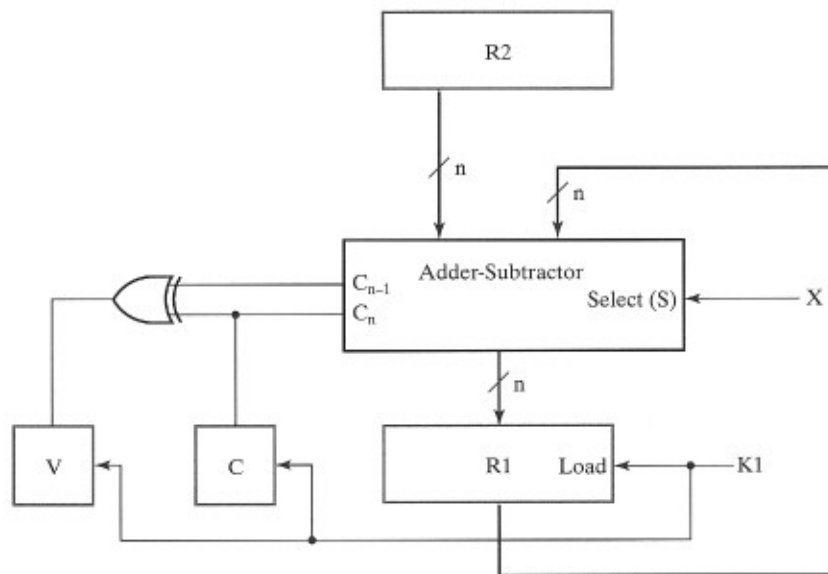
Multiplication and division are not listed in Table 7-3. Multiplication can be represented by the symbol $*$ and division by $/$. These two operations are not included in the basic set of arithmetic microoperations because they are assumed to be implemented by sequences of basic microoperations. In contrast, multiplication can be considered as a microoperation if implemented by a combinational circuit as illustrated in Section 5-4. In such a case, the result is transferred into a destination register at the clock edge after all signals have propagated through the entire combinational circuit.

There is a direct relationship between the statements written in register transfer notation and the registers and digital functions required for their implementation. To illustrate, consider the following two statements:

$$\overline{X}K_1: R1 \leftarrow R1 + R2$$

$$XK_1: R1 \leftarrow R1 + \overline{R2} + 1$$

Control variable K_1 activates an operation to add or subtract. If, at the same time, control variable X is equal to 0, then $\overline{X}K_1 = 1$, and the contents of $R2$ are added to the contents of $R1$. If X is equal to 1, then $XK_1 = 1$, and the contents of $R2$ are subtracted from the contents of $R1$. Note that the two control conditions are Boolean functions and reduce to 0 when $K_1 = 0$, a condition that inhibits the execution of both operations simultaneously.



□ **FIGURE 7-6**
Implementation of Add and Subtract Microoperations

A block diagram showing the implementation of the preceding two statements is given in Figure 7-6. An n -bit adder-subtractor, similar to the one shown in Figure 5-8, receives its input data from registers $R1$ and $R2$. The sum or difference is applied to the inputs of $R1$. The Select input S of the adder-subtractor selects the operation in the circuit. When $S = 0$, the two inputs are added, and when $S = 1$, $R2$ is subtracted from $R1$. Applying the control variable X to the S input activates the required operation. The output of the adder-subtractor is loaded into $R1$ on any positive clock edge at which $\bar{X}K_1 = 1$ or $XK_1 = 1$. We can simplify this to just K_1 , since

$$\bar{X}K_1 + XK_1 = (\bar{X} + X)K_1 = K_1$$

Thus, the control variable X selects the operation, and the control variable K_1 loads the result into $R1$.

Based on the discussion of overflow in Section 5-3, the overflow output is transferred to flip-flop V , and the output carry from the most significant bit of the adder-subtractor is transferred to flip-flop C , as shown in Figure 7-6. These transfers occur when $K_1 = 1$ and are not represented in the register transfer statements; if desired, we could show them as additional simultaneous transfers.

Logic Microoperations

Logic microoperations are useful in manipulating the bits stored in a register. These operations consider each bit in the register separately and treat it as a binary variable. The symbols for the four basic logic operations are shown in Table 7-4. The NOT microoperation, represented by a bar over the source register name, complements all bits and thus is the same as the 1's complement. The symbol \wedge is used to denote the AND microoperation and the symbol \vee to denote the OR microoperation. By using these special symbols, it is possible to distinguish between the add microoperation represented by a $+$ and the OR microoperation. Although the $+$ symbol has two meanings, one can distinguish between them by noting where the symbol occurs. If the $+$ occurs in a microoperation, it denotes addition. If the $+$ occurs in a control or Boolean function, it denotes OR. The OR microoperation will always use the \vee symbol. For example, in the statement

$$(K_1 + K_2): R1 \leftarrow R2 + R3, R4 \leftarrow R5 \vee R6$$

□ TABLE 7-4
Logic Microoperations

Symbolic designation	Description
$R0 \leftarrow \bar{R1}$	Logical bitwise NOT (1's complement)
$R0 \leftarrow R1 \wedge R2$	Logical bitwise AND (clears bits)
$R0 \leftarrow R1 \vee R2$	Logical bitwise OR (sets bits)
$R0 \leftarrow R1 \oplus R2$	Logical bitwise XOR (complements bits)

The $+$ between R_1 and R_2 is an OR operation between two variables in a control condition. The $+$ between R_2 and R_3 specifies an add microoperation. The OR microoperation is designated by the symbol \vee between registers R_5 and R_6 . The logic microoperations can be easily implemented with a group of gates, one for each bit position. The NOT of a register of n bits is obtained with n NOT gates in parallel. The AND microoperation is obtained using a group of n AND gates, each of which receives a pair of corresponding inputs from the two source registers. The outputs of the AND gates are applied to the corresponding inputs of the destination register. The OR and exclusive-OR microoperations require a similar arrangement of gates.

The logic microoperations can change bit values, clear a group of bits, or insert new bit values into a register. The following examples show how the bits stored in the 16-bit register R_1 can be selectively changed by using a logic microoperation and a logic operand stored in the 16-bit register R_2 .

The AND microoperation can be used for clearing one or more bits in a register to 0. The Boolean equations $X \cdot 0 = 0$ and $X \cdot 1 = X$ dictate that, when ANDed with 0, a binary variable X produces a 0, but when ANDed with 1, the variable remains unchanged. A given bit or group of bits in a register can be cleared to 0 if ANDed with 0. Consider the following example:

10101101 10101011	R_1	(data)
00000000 11111111	R_2	(mask)
00000000 10101011	$R_1 \leftarrow R_1 \wedge R_2$	

The 16-bit logic operand in R_2 has 0's in the high-order byte and 1's in the low-order byte. By ANDing the contents of R_2 with the contents of R_1 , it is possible to clear the high-order byte of R_1 and leave the bits in the low-order byte unchanged. Thus, the AND operation can be used to selectively clear bits of a register. This operation is sometimes called *masking out* the bits, because it masks or deletes all 1's in the *data* in R_1 , based on bit positions that are 0 in the *mask* provided in R_2 .

The OR microoperation is used to set one or more bits in a register. The Boolean equations $X + 1 = 1$ and $X + 0 = X$ dictates that, when ORed with 1, the binary variable X produces a 1, but when ORed with 0, the variable remains unchanged. A given bit or group of bits in a register can be set to 1 if ORed with 1. Consider the following example:

10101101 10101011	R_1	(data)
11111111 00000000	R_2	(mask)
11111111 10101011	$R_1 \leftarrow R_1 \vee R_2$	

The high-order byte of R_1 is set to all 1's by ORing it with all 1's in the R_2 operand. The low-order byte remains unchanged because it is ORed with 0's.

The XOR (exclusive-OR) microoperation can be used to complement one or more bits in a register. The Boolean equations $X \oplus 1 = \bar{X}$ and $X \oplus 0 = X$ dictate

that, when a binary variable X is XORed with 1, it is complemented, but when XORed with 0, the variable remains unchanged. By XORing a bit or group of bits in register $R1$ with 1's in selected positions in $R2$, it is possible to complement the bits in the selected positions in $R1$. Consider the following example:

10101101 10101011	$R1$	(data)
11111111 00000000	$R2$	(mask)
01010010 10101011	$R1 \leftarrow R1 \oplus R2$	

The high-order byte in $R1$ is complemented after the XOR operation with $R2$, and the low-order byte is unchanged.

Shift Microoperations

Shift microoperations are used for lateral movement of data. The contents of a source register can be shifted either right or left. A *left shift* is toward the most significant bit, and a *right shift* is toward the least significant bit. Shift microoperations are used in the serial transfer of data. They are also used for manipulating the contents of registers in arithmetic, logical, and control operations. The destination register for a shift microoperation may be the same as or different from the source register. We use strings of letters to represent the shift microoperations defined in Table 7-5. For example,

$$R0 \leftarrow sr\ R0, R1 \leftarrow sl\ R2$$

are two microoperations that respectively specify a 1-bit shift to the right of the contents of register $R0$ and a transfer of the contents of $R2$ shifted one bit to the left into register $R1$. The contents of $R2$ are not changed by these shifts.

For a left-shift microoperation, we call the rightmost bit of the destination register the *incoming bit*. For a right-shift microoperation, we define the leftmost bit of the destination register as the incoming bit. The incoming bit may have different values, depending upon the type of shift microoperation. Here we assume that, for sr and sl , the incoming bit is 0, as shown in the examples in Table 7-5. The *outgoing bit* is the leftmost bit of the source register for the left-shift operation and the rightmost bit of the source register for the right-shift operation. For the left and right shifts shown, the outgoing bit value is simply discarded. In chapter 11, we will

□ **TABLE 7-5**
Examples of Shifts

Type	Symbolic designation	Eight-bit examples	
		Source $R2$	After shift: Destination $R1$
shift left	$R1 \leftarrow sl\ R2$	10011110	00111100
shift right	$R1 \leftarrow sr\ R2$	11100101	01110010

explore other types of shifts differing in the way incoming and outgoing bits are treated.

7-6 MICROOPERATIONS ON A SINGLE REGISTER

This section covers the implementation of one or more microoperations with a single register as the destination of all primary results. The single register may also serve as a source of an operand for binary and unary operations. Due to the close ties between a single set of storage elements and the microoperations, the combinational logic implementing the microoperations is assumed to be a part of the register and is called *dedicated logic* of the register. This is in contrast to logic which is shared by multiple destination registers. In this case, the combinational logic implementing the microoperations is called *shared logic* for the set of destination registers.

The combinational logic implementing the microoperations described in the previous section can use one or more functional blocks from chapters 4 and 5 or can be designed specifically for the register. Initially, functional blocks will be used in combination with D flip-flops or D-flip-flops with enable. A simple technique using multiplexers for selection is introduced to allow multiple microoperations on a single register. Next, single and multiple function registers that perform shifting and counting are designed.

Multiplexer-Based Transfers

There are occasions when a register receives data from two or more different sources at different times. Consider the following conditional statement having an *if-then-else* form:

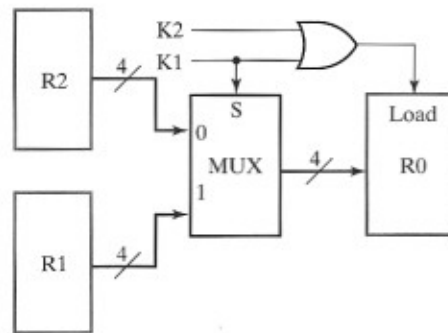
$$\text{if}(K_1 = 1)\text{then}(R0 \leftarrow R1) \text{ else if } (K_2 = 1) \text{ then } (R0 \leftarrow R2)$$

The value in register $R1$ is transferred to register $R0$ when control signal K_1 equals 1. When $K_1 = 0$, the value in register $R2$ is transferred to $R0$ when K_2 equals 1. Otherwise, the contents of $R0$ remains unchanged. The conditional statement may be broken into two parts using the following control conditions:

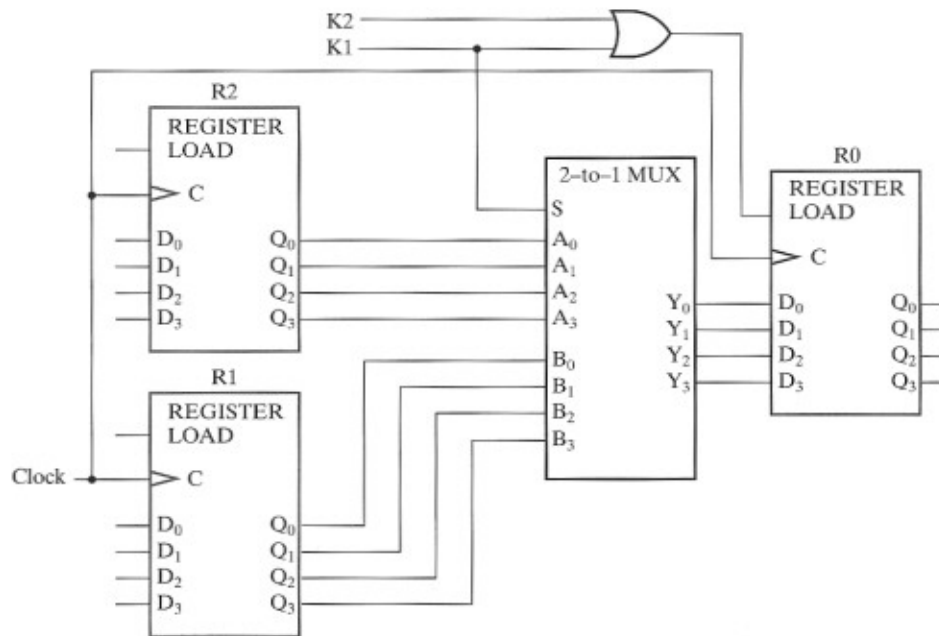
$$K_1: R0 \leftarrow R1, \bar{K}_1 K_2: R0 \leftarrow R2$$

This specifies hardware connections from two registers, $R1$ and $R2$, to one common destination register $R0$. In addition, making a selection between two source registers must be based on values of the control variables K_1 and K_2 .

The block diagram for a circuit with 4-bit registers that implements the conditional register transfer statements using a multiplexer is shown in Figure 7-7(a). The quad 2-to-1 multiplexer selects between the two source registers. For $K_1 = 1$, $R1$ is loaded into $R0$, irrespective of the value of K_2 . For $K_1 = 0$ and $K_2 = 1$, $R2$ is loaded into $R0$. When both K_1 and K_2 are equal to 0, the multiplexer selects $R2$ as the input to $R0$, but, because the control function, $K_2 + K_1$, connected to the LOAD input of $R0$ equals 0, the contents of $R0$ remain unchanged.



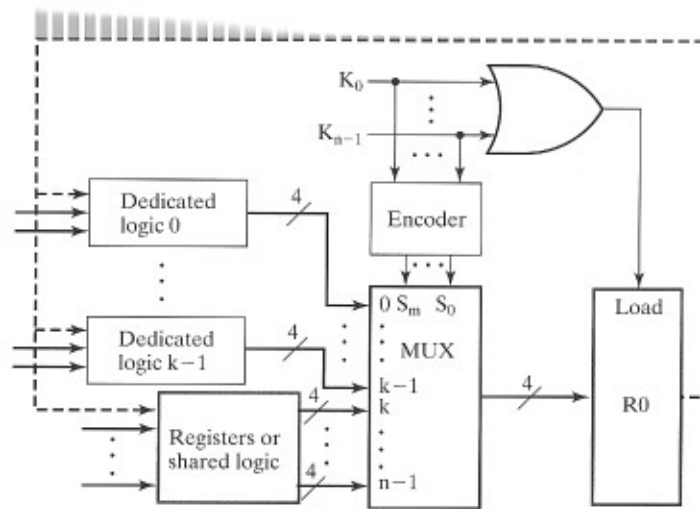
(a) Block diagram



(b) Detailed logic

□ **FIGURE 7-7**
Use of Multiplexers to Select between Two Registers

The detailed logic diagram for the hardware implementation is shown in Figure 7-7(b). The diagram uses functional block symbols based upon detailed logic for the registers in Figure 7-2 and for a quad 2-to-1 multiplexer from Chapter 4. Note that since this diagram represents just a part of a system, there are inputs and outputs that are not yet connected. Also, the clock is not shown in the block diagram, but is shown in the detailed diagram. It is important to relate the information given in a block diagram such as Figure 7-7(a) with the detailed wiring connections in the corresponding logic diagram in Figure 7-7(b). In order



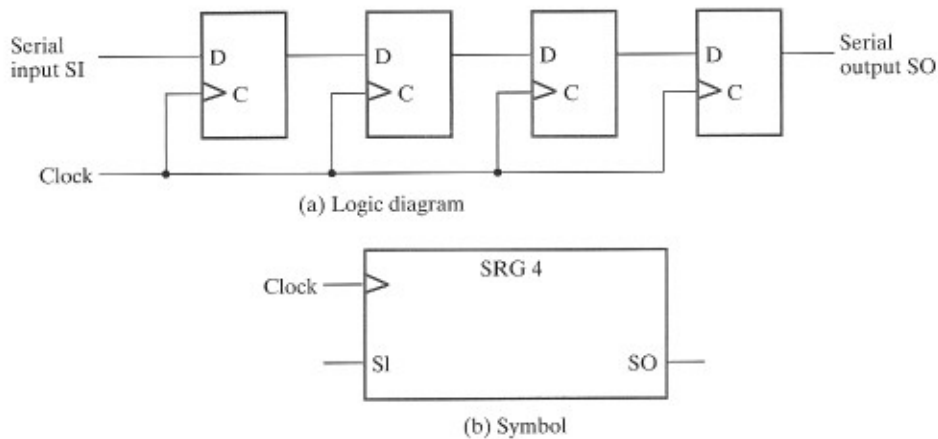
□ **FIGURE 7-8**
Generalization of Multiplexer Selection for n Sources

to save space, we often omit the detailed logic diagrams in designs. However, it is possible to obtain a logic diagram with detailed wiring from the corresponding block diagram and a library of functional blocks. In fact, such a procedure is performed by computer programs used for automated logic synthesis.

The preceding example can be generalized by allowing the multiplexer to have n sources and these sources to be register outputs or combinational logic implementing microoperations. This generalization results in the block diagram shown in Figure 7-8. The diagram assumes that each source is either the outputs of a register or of combinational logic implementing one or more microinstructions. In those cases in which the microoperations are dedicated to the register, the corresponding dedicated logic is included as a part of the register. In Figure 7-8, the first k sources are dedicated logic and the last $n - k$ sources are either registers or shared logic. The control signals that select a given source are either a single control variable or the OR of all control signals corresponding to the microoperations associated with the source. To force $R0$ to load for a microoperation, these control signals are ORed together to form the *Load* signal. Since it is assumed that only one of the control signals is 1 at any time, these signals must be encoded to provide the selection codes for the multiplexer. Two modifications to the given structure are possible. The control signals could be applied directly to a $2 \times n$ AND-OR circuit (i.e., a multiplexer with the decoder deleted). Alternatively, the control signals could already be encoded, omitting the use of the all-zero code, so that the OR gate still forms the *Load* signal correctly.

Shift Registers

A register capable of shifting its stored bits laterally in one or both directions is called a *shift register*. The logical configuration of a shift register consists of a



□ **FIGURE 7-9**
4-Bit Shift Register

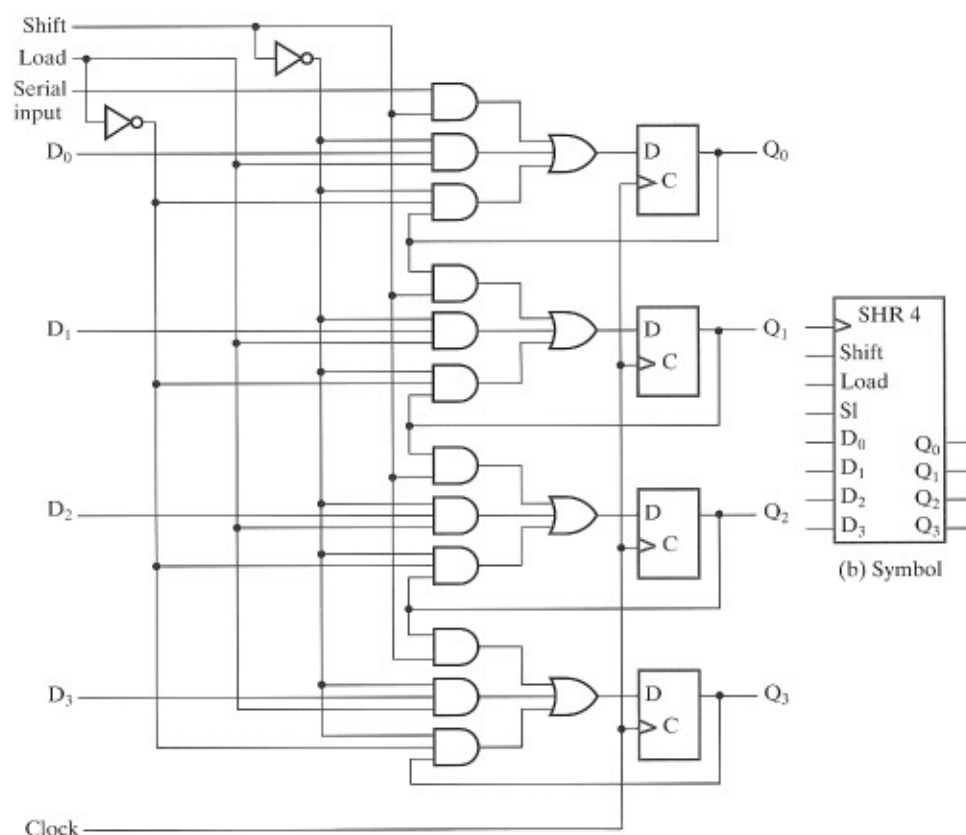
chain of flip-flops, with the output of one flip-flop connected to the input of the next flip-flop. All flip-flops have a common clock pulse input that activates the shift.

The simplest possible shift register uses only flip-flops, as shown in Figure 7-9(a). The output of a given flip-flop is connected to the *D* input of the flip-flop at its right. The clock is common to all flip-flops. The *serial input SI* is the input to the leftmost flip-flop. The *serial output SO* is taken from the output of the rightmost flip-flop. A symbol for the shift register is given in Figure 7-9(b).

Sometimes it is necessary to control the register so that it shifts only on select positive clock edges. For the shift register in Figure 7-9, the shift can be controlled by connecting the clock through the logic shown in Figure 7-1(c), with *Shift* replacing *Load*. Again, due to clock skew, this is usually not the most desirable approach. Thus, we learn later that the shift operation can be controlled through the *D* inputs of the flip-flops rather than through the clock inputs *C*.

SHIFT REGISTER WITH PARALLEL LOAD If all flip-flop outputs of a shift register are accessible, then information entered serially by shifting can be taken out in parallel from the flip-flop outputs. If a parallel load capability is also added to a shift register, then data entered in parallel can be shifted out serially. Thus, a shift register with accessible flip-flop outputs and parallel load can be used for converting incoming parallel data to outgoing serial data and vice versa.

The logic diagram for a 4-bit shift register with parallel load and the symbol for this register are shown in Figure 7-10. There are two control inputs, one for the shift and the other for the load. Each stage of the register consists of a *D* flip-flop, an OR gate, and three AND gates. The first AND gate enables the shift operation. The second AND gate enables the input data. The third AND gate restores the contents of the register when no operation is required.



□ **FIGURE 7-10**
Shift Register with Parallel Load

The operation of this register is specified in Table 7-6 and is also given by the register transfers:

$$\begin{aligned} \text{Shift: } Q &\leftarrow \text{sl}Q \\ \overline{\text{Shift}} \cdot \text{Load: } Q &\leftarrow D \end{aligned}$$

The “No Change” operation is implicit if neither of the conditions for transfers is satisfied. When both the shift and load control inputs are 0, the third AND gate in each stage is enabled, and the output of each flip-flop is applied to its own D input. A positive transition of the clock restores the contents to the register, and the output is unchanged. When the shift input is 0 and the load input is 1, the second AND gate in each stage is enabled, and the input D_i is applied to the D input of the corresponding flip-flop. The next positive clock transition transfers the parallel input data into the register. When the shift input is equal to 1, the first AND gate in each

□ **TABLE 7-6**
Function Table for the Register of Figure 7-10

Shift	Load	Operation
0	0	No change
0	1	Load parallel data
1	×	Shift down from Q_0 to Q_3

stage is enabled and the other two are disabled. Since the Load input is disabled by the Shift input on the second AND gate, we mark it with a don't-care condition in the Shift row of the table. When a positive edge occurs on the clock, the shift operation causes the data from the serial input SI to be transferred to flip-flop Q_0 , the output of Q_0 to be transferred to flip-flop Q_1 , and so on down the line. Note that because of the way the circuit is drawn, the shift occurs in the downward direction. If we rotate the page a quarter turn counterclockwise, the register shifts from left to right.

Shift registers are often used to interface digital systems that are distant from each other. For example, suppose it is necessary to transmit an n -bit quantity between two points. If the distance is far, it will be expensive to use n lines to transmit the n bits in parallel. It may be more economical to use a single line and transmit the information serially, one bit at a time. The transmitter loads the n -bit data in parallel into a shift register and then transmits the data serially along the common line. The receiver accepts the data serially into a shift register. When all n bits are accumulated, they can be taken in parallel from the outputs of the register. Thus, the transmitter performs a parallel-to-serial conversion of data, and the receiver does a serial-to-parallel conversion.

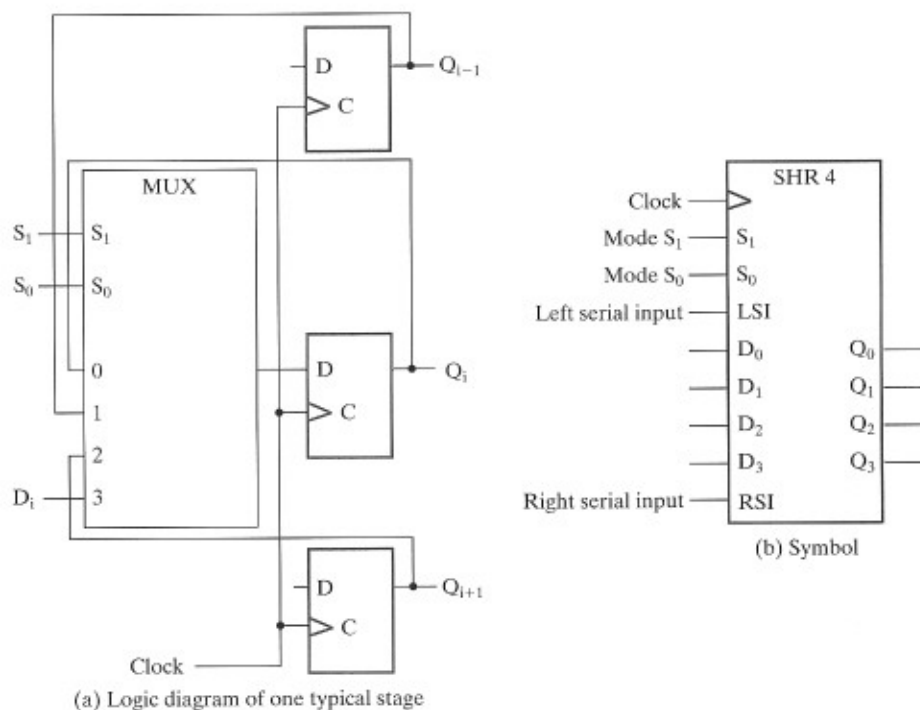
BIDIRECTIONAL SHIFT REGISTER A register capable of shifting in only one direction is called a *unidirectional shift register*. A register that can shift in both directions is called a *bidirectional shift register*. It is possible to modify the circuit of Figure 7-10, by adding a fourth AND gate in each stage, for shifting the data in the upward direction. An investigation of the resultant circuit will reveal that the four AND gates, together with the OR gate in each stage, constitute a multiplexer with the selection inputs controlling the operation of the register.

One stage of a bidirectional shift register with parallel load is shown in Figure 7-11(a). Each stage consists of a D flip-flop and a 4-to-1-line multiplexer. The two selection inputs S_1 and S_0 select one of the multiplexer inputs to apply to the D flip-flop. The selection lines control the mode of operation of the register according to the function table of Table 7-7 and the register transfers:

$$\bar{S}_1 \cdot S_0: Q \leftarrow slQ$$

$$S_1 \cdot \bar{S}_0: Q \leftarrow srQ$$

$$S_1 \cdot S_0: Q \leftarrow D$$



□ **FIGURE 7-11**
Bidirectional Shift Register with Parallel Load

□ **TABLE 7-7**
Function Table for the Register of Figure 7-7

Mode control		Register Operation
S_1	S_0	
0	0	No change
0	1	Shift down
1	0	Shift up
1	1	Parallel load

The “No Change” operation is implicit if none of the conditions for transfers is satisfied. When the mode control $S_1S_0 = 00$, input 0 of the multiplexer is selected. This forms a path from the output of each flip-flop into its own input. The next clock transition transfers the current stored value back into each flip-flop, and no change of state occurs. When $S_1S_0 = 01$, the terminal marked 1 on the multiplexer has a path to the D input of each flip-flop. These paths cause a shift-down operation. The serial input is transferred into the first stage, and the content of each stage Q_{i-1} , is

transferred into stage Q_i . When $S_1S_0 = 10$, a shift-up operation results in a second serial input that enters the last stage. In addition, the value in each stage Q_{i+1} is transferred into stage Q_i . Finally, when $S_1S_0 = 11$, the binary information on each parallel input line is transferred into the corresponding flip-flop, resulting in a parallel load.

Figure 7-11(b) shows a symbol for the bidirectional shift register from Figure 7-11(a). Note that both a left serial input (LSI) and a right serial input (RSI) are provided. If serial outputs are desired, Q_3 is used for left shift and Q_0 for right shift.

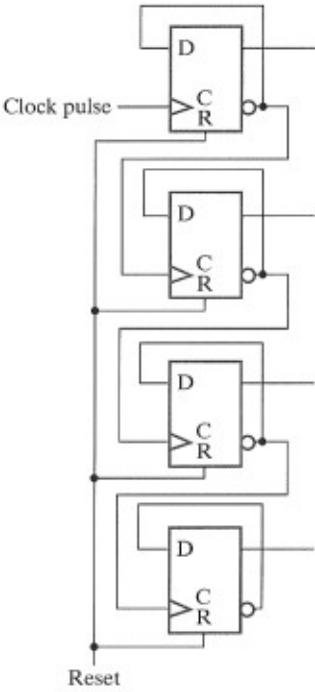
Ripple Counter

A register that goes through a prescribed sequence of distinct states upon the application of a sequence of input pulses is called a *counter*. The input pulses may be clock pulses or may originate from some other source, and they may occur at regular or irregular intervals of time. In our discussion of counters, we assume clock pulses, but other signals can be substituted for the clock. The sequence of states may follow the binary number sequence or any other prescribed sequence of states. A counter that follows the binary number sequence is called a *binary counter*. An n -bit binary counter consists of n flip-flops and can count in binary from 0 through $2^n - 1$.

Counters are available in two categories: ripple counters and synchronous counters. In a ripple counter, the flip-flop output transitions serve as the sources for triggering the changes in other flip-flops. In other words, the C inputs of some of the flip-flops are triggered not by the common clock pulse, but rather by the transitions that occur on other flip-flop outputs. In a synchronous counter, the C inputs of all flip-flops receive the common clock pulse, and the change of state is determined from the present state of the counter. Synchronous counters are discussed in the next two subsections. Here we present the binary ripple counter and explain its operation.

The logic diagram of a 4-bit binary ripple counter is shown in Figure 7-12. The counter is constructed from D flip-flops connected such that the application of a positive edge to the C input of each flip-flop causes the flip-flop to complement its state. The complemented output of each flip-flop is connected to the C input of the next most significant flip-flop. The flip-flop holding the least significant bit receives the incoming clock pulses. Positive-edge triggering makes each flip-flop complement its value when the signal on its C input goes through a positive transition. The positive transition occurs when the complemented output of the previous flip-flop, to which C is connected, goes from 0 to 1. A 1-level signal on *Reset* driving the R inputs clears the register to all zeros asynchronously.

To understand the operation of a binary ripple counter, let us examine the upward counting sequence given in the left half of Table 7-8. The count starts at binary 0 and increments by one with each count pulse. After the count of 15, the counter goes back to 0 to repeat the count. The least significant bit (Q_0) is complemented by each count pulse. Every time that Q_0 goes from 1 to 0, $\overline{Q_0}$ goes from 0 to 1, complementing Q_1 . Every time that Q_1 goes from 1 to 0, it complements Q_2 . Every time that Q_2 goes



□ FIGURE 7-12
4-Bit Ripple Counter

□ TABLE 7-8
Counting Sequence of Binary Counter

Upward Counting Sequence				Downward Counting Sequence			
Q ₃	Q ₂	Q ₁	Q ₀	Q ₃	Q ₂	Q ₁	Q ₀
0	0	0	0	1	1	1	1
0	0	0	1	1	1	1	0
0	0	1	0	1	1	0	1
0	0	1	1	1	1	0	0
0	1	0	0	1	0	1	1
0	1	0	1	1	0	1	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	0	0
1	0	0	0	0	1	1	1
1	0	0	1	0	1	1	0
1	0	1	0	0	1	0	1
1	0	1	1	0	1	0	0
1	1	0	0	0	0	1	1
1	1	0	1	0	0	1	0
1	1	1	0	0	0	0	1
1	1	1	1	0	0	0	0

from 1 to 0, it complements Q_3 , and so on for any higher order bits in the ripple counter. For example, consider the transition from count 0011 to 0100. Q_0 is complemented with the count pulse positive edge. Since Q_0 goes from 1 to 0, it triggers Q_1 and complements it. As a result, Q_1 goes from 1 to 0, which complements Q_2 , changing it from 0 to 1. Q_2 does not trigger Q_3 , because $\overline{Q_2}$ produces a negative transition, and the flip-flops respond only to positive transitions. Thus, the count from 0011 to 0100 is achieved by changing the bits one at a time. The counter goes from 0011 to 0010 (Q_0 from 1 to 0), then to 0000 (Q_1 from 1 to 0), and finally to 0100 (Q_2 from 0 to 1). The flip-flops change one at a time in quick succession as the signal propagates through the counter in a ripple fashion from one stage to the next.

A ripple counter that counts downward gives the sequence in the right half of Table 7-8. Downward counting can be accomplished by connecting the true output of each flip-flop to the C input of the next flip-flop.

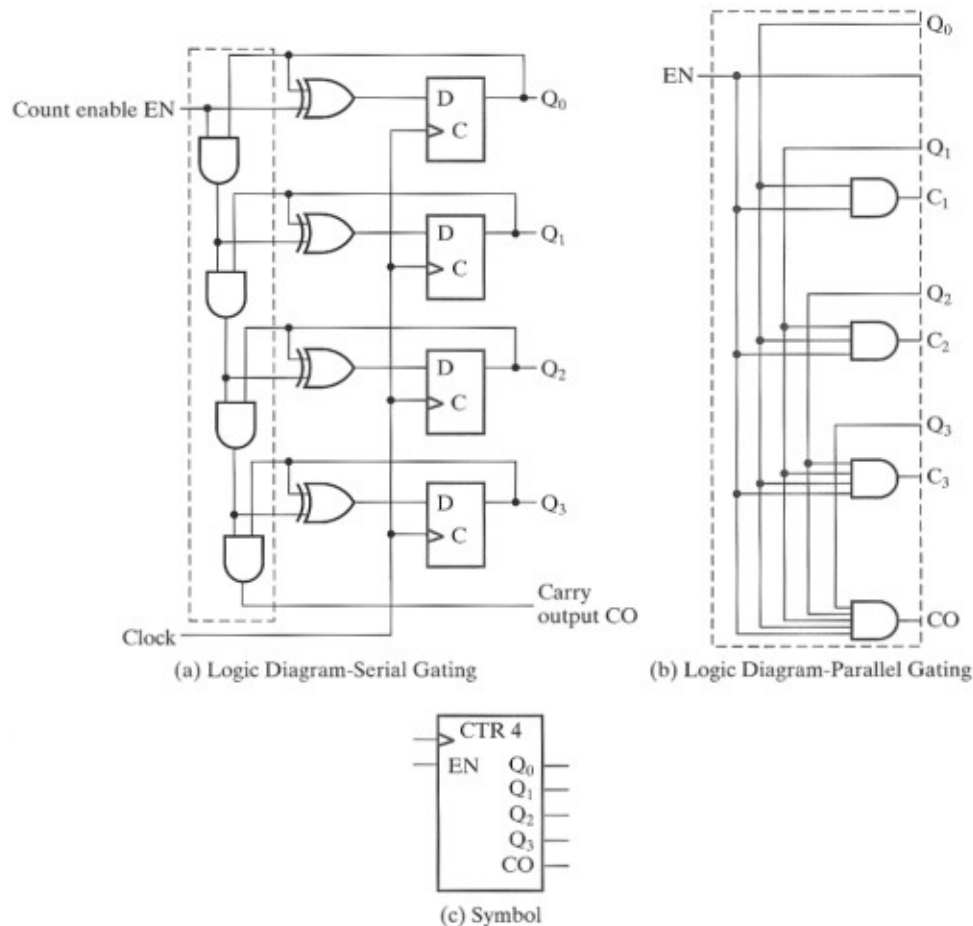
The advantage of ripple counters is their simple hardware. Unfortunately, they are asynchronous circuits and, with added logic, can become circuits with delay dependence and unreliable operation. This is particularly true for logic that provides feedback paths from counter outputs back to counter inputs. Also, due to the length of time required for the ripple to finish, large ripple counters can be slow circuits. As a consequence, synchronous binary counters are favored in all but low-power designs where ripple counters have an advantage. (See Problem 7-11.)

Synchronous Binary Counters

Synchronous counters, in contrast to ripple counters, have the clock applied to the C inputs of all flip-flops. Thus, the common clock pulse triggers all flip-flops simultaneously rather than one at a time, as in a ripple counter. A synchronous binary counter that counts up by 1 can be constructed from the incrementer in Figure 5-12 and D flip-flops as shown in Figure 7-13(a). The carry output CO is added by not placing an X value on the C_4 output before the contraction of an adder to the incrementer in Figure 5-12. Output CO is used to extend the counter to more stages.

Note that the flip-flops trigger on the positive-edge transition of the clock. The polarity of the clock is not essential here, like it was for the ripple counter. The synchronous counter can be triggered with either the positive or the negative clock transition.

SERIAL AND PARALLEL COUNTERS We will use the synchronous counter in Figure 7-13 to demonstrate two alternative designs for binary counters. In Figure 7-13(a), a chain of 2-input AND gates is used to provide information to each stage about the state of the prior stages in the counter. This is analogous to the carry logic in the ripple carry adder. A counter that uses such logic is said to have *serial gating* and is referred to as a *serial counter*. The analogy to the ripple carry adder suggests that there might be counter logic analogous to the carry lookahead adder. Such logic can be derived by contracting a carry lookahead adder, with the result shown in Figure 7-13(b). This logic can simply replace that in the blue box in Figure 7-13(a) to produce a counter with *parallel gating*, called a *parallel counter*.



□ **FIGURE 7-13**
4-Bit Synchronous Binary Counter

The advantage of parallel gating logic is that, in going from state 1111 to state 0000, only one AND gate delay occurs instead of the four AND gate delays that occur for the serial counter. This reduction in delay allows the counter to operate much faster.

If we connect two 4-bit parallel counters together by connecting the *CO* output of one to the *EN* input of the other, the result is an 8-bit serial-parallel counter. This counter has two 4-bit parallel parts connected in series with each other. The idea can be extended to counters of any length. Again, employing the analogy to carry lookahead adders, additional levels of gating logic can be introduced to replace the serial connections between the 4-bit segments. The added reduction in delay that results is useful for constructing large, fast counters.

The symbol for the 4-bit counter using positive-edge triggering is shown in Figure 7-13(c).

UP-DOWN BINARY COUNTER A synchronous count-down binary counter goes through the binary states in reverse order from 1111 to 0000 and back to 1111 to repeat the count. The logic diagram of a synchronous count-down binary counter is similar to the circuit for the binary up-counter, except that a decrementer is used instead of an incrementer. The two operations can be combined to form a counter that can count both up and down, which is referred to as an up-down binary counter. Such a counter can be designed by contracting the adder-subtractor in Figure 5-8 into an incrementer-decrementer and adding the D flip-flops. The counter counts up for $S = 0$ and down for $S = 1$.

Alternatively, an up-down counter with ENABLE can be designed directly from counter behavior. It needs a mode input to select between the two operations. We designate this mode select input by S , with $S = 0$ for up-counting and $S = 1$ for down-counting. Let variable EN be a count enable input, with $EN = 1$ for normal up- or down-counting and $EN = 0$ for disabling both counts. A 4-bit up-down binary counter can be described by the following flip-flop input equations:

$$D_{A0} = Q_0 \oplus EN$$

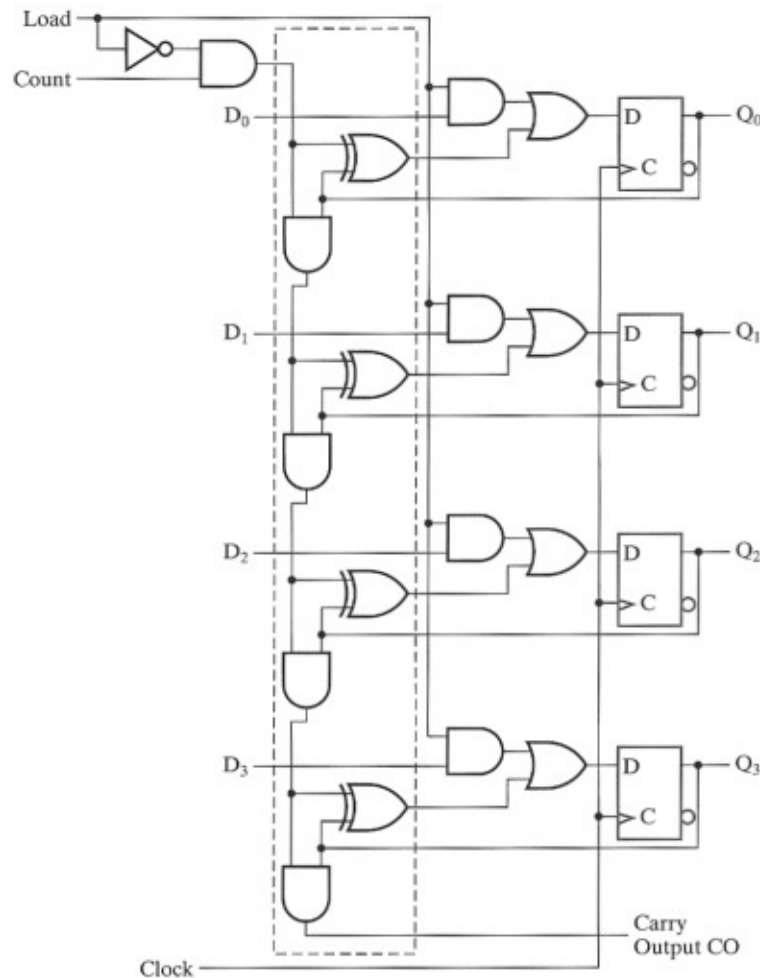
$$D_{A1} = Q_1 \oplus ((Q_0 \cdot \bar{S} + \bar{Q}_0 \cdot S) \cdot EN)$$

$$D_{A2} = Q_2 \oplus ((Q_0 \cdot Q_1 \cdot \bar{S} + \bar{Q}_0 \cdot \bar{Q}_1 \cdot S) \cdot EN)$$

$$D_{A3} = Q_3 \oplus ((Q_0 \cdot Q_1 \cdot Q_2 \cdot \bar{S} + \bar{Q}_0 \cdot \bar{Q}_1 \cdot \bar{Q}_2 \cdot S) \cdot EN)$$

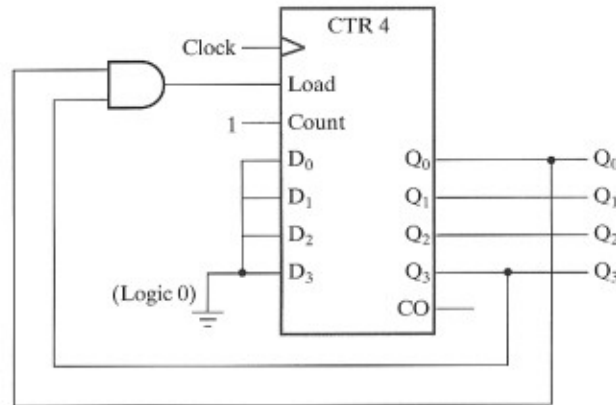
The logic diagram of the circuit can be easily obtained from the input equations, but is not included here. It should be noted that the equations, as written, provide parallel gating using distinct carry logic for up-counting and down-counting. It is also possible to use two distinct serial gating chains as well. In contrast, the counter derived using the incrementer-decrementer uses only a single carry chain. Overall, the logic cost is similar.

BINARY COUNTER WITH PARALLEL LOAD Counters employed in digital systems quite often require a parallel-load capability for transferring an initial binary number into the counter prior to the count operation. This function can be implemented by an incrementer with an ENABLE, n ENABLEs, and n 2-input OR gates as shown in Figure 7-14. The n ENABLEs are used to enable and disable the parallel load of input data, D , using the signal Load. Note that ENABLE on the incrementer is used to enable or disable counting using Count Load. With the Load and Count inputs both at 0, the outputs do not change, even when pulses are applied to the C inputs. If the load input is maintained at logic 0, the Count input controls the operation of the counter, and the outputs change to the next binary count for each positive transition of the clock. The data applied to the D inputs is loaded into the flip-flops when Load equals 1, regardless of the value of Count because Load is ANDed with Count. Counters with parallel load are very useful in the design of digital computers. In subsequent chapters, we will refer to them as registers with load and increment operations.



□ **FIGURE 7-14**
4-Bit Binary Counter with Parallel Load

The binary counter with parallel load can be converted into a synchronous BCD counter (without load input) by connecting an external AND gate to it, as shown in Figure 7-15. The counter starts with an all-zero output, and the count input is always active. As long as the output of the AND gate is 0, each positive clock edge increments the counter by one. When the output reaches the count of 1001, both Q_0 and Q_3 become 1, making the output of the AND gate equal to 1. This condition makes *Load* active; so on the next clock transition, the counter does not count, but is loaded from its four inputs. Since all four inputs are connected to logic 0, 0000 is loaded into the counter following the count of 1001. Thus, the circuit counts from 0000 through 1001, followed by 0000, as required for a BCD counter.



□ **FIGURE 7-15**
BCD Counter

Other Counters

Counters can be designed to generate any desired number of states in sequence. A *divide-by-N counter* (also known as a *modulo-N counter*) is a counter that goes through a repeated sequence of N states. The sequence may follow the binary count or may be any other arbitrary sequence. In either case, the design of the counter follows the procedure presented in Chapter 6 for the design of synchronous sequential circuits. To demonstrate this procedure, we will present the design of two counters: a BCD counter and a counter with an arbitrary sequence of states.

BCD COUNTER As shown in the previous section, a BCD counter can be obtained from a binary counter with parallel load. It is also possible to design a BCD counter directly using individual flip-flops and gates. Assuming D -type flip-flops for the counter, we list the present states and corresponding next states in Table 7-9. An output Y is included in the table. This output is equal to 1 when the present state is 1001. In this way, Y can enable the count of the next decade while its own decade switches from 1001 to 0000.

The flip-flop input equations for D are obtained from the next-state values listed in the table and can be simplified by means of K-maps. The unused states for minterms 1010 through 1111 are used as don't-care conditions. The simplified input equations for the BCD counter are

$$\begin{aligned} D_1 &= \overline{Q}_1 \\ D_2 &= Q_2 \oplus Q_1 \overline{Q}_8 \\ D_4 &= Q_4 \oplus Q_1 Q_2 \\ D_8 &= Q_8 \oplus (Q_1 Q_8 + Q_1 Q_2 Q_4) \\ Y &= Q_1 Q_8 \end{aligned}$$

□ **TABLE 7-9**
State Table and Flip-Flop Inputs for BCD Counter

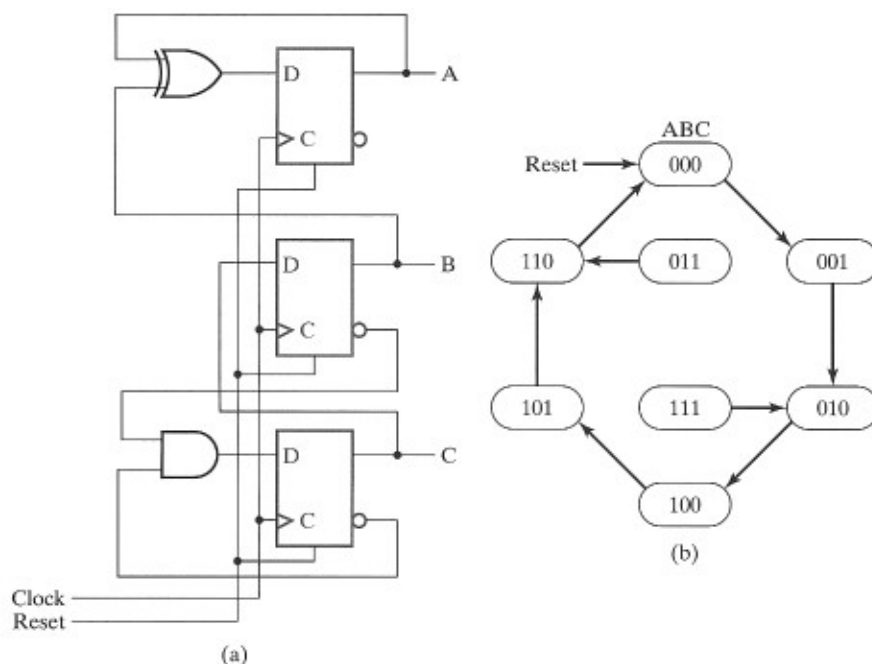
Present State				Next State				Output
Q_8	Q_4	Q_2	Q_1	$D_8 = Q_8(t+1)$	$D_4 = Q_4(t+1)$	$D_2 = Q_2(t+1)$	$D_1 = Q_1(t+1)$	Y
0	0	0	0	0	0	0	1	0
0	0	0	1	0	0	1	0	0
0	0	1	0	0	0	1	1	0
0	0	1	1	0	1	0	0	0
0	1	0	0	0	1	0	1	0
0	1	0	1	0	1	1	0	0
0	1	1	0	0	1	1	1	0
0	1	1	1	1	0	0	0	0
1	0	0	0	1	0	0	1	0
1	0	0	1	0	0	0	0	1

□ **TABLE 7-10**
State Table and Flip-Flop Inputs for Counter

Present State			Next State		
A	B	C	$DA = A(t+1)$	$DB = B(t+1)$	$DC = C(t+1)$
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	0	0	0

Synchronous BCD counters can be cascaded to form counters for decimal numbers of any length. The cascading is done by replacing D_1 with $D_1 = Q_1 \oplus Y$ where Y is from the next lower BCD counter. Also, Y needs to be ANDed with the product terms to the right of each of the XOR symbol in each of the equations for D_2 through D_8 .

ARBITRARY COUNT SEQUENCE Suppose we wish to design a counter that has a repeated sequence of six states, as listed in Table 7-10. In this sequence, flip-flops B and C repeat the binary count 00, 01, 10, while flip-flop A alternates between 0 and 1 every three counts. Thus, the count sequence for the counter is not straight binary, and two states, 011 and 111, are not included in the count. The D flip-flop



□ **FIGURE 7-16**
Counter with Arbitrary Count

input equations can be simplified using minterms 3 and 7 as don't-care conditions. The simplified functions are

$$\begin{aligned} D_A &= A \oplus B \\ D_B &= C \\ D_C &= \bar{B} \bar{C} \end{aligned}$$

The logic diagram of the counter is shown in Figure 7-16(a). Since there are two unused states, we analyze the circuit to determine their effect. The state diagram obtained is drawn in Figure 7-16(b). This diagram indicates that if the circuit ever goes to one of the unused states, the next count pulse transfers it to one of the valid states, and the circuit then continues to count correctly.

7-7 REGISTER CELL DESIGN

In Section 5-1, we discussed iterative combinational circuits. In this chapter, we connect such circuits to flip-flops to form sequential circuits. A single-bit cell of an iterative combinational circuit connected to a flip-flop that provides the output forms a two-state sequential circuit called a *register cell*. We can design an n -bit register with one or more associated microoperations by designing a register cell and making n copies of it. Depending on whether the output of the flip-flop is an input to the iterative circuit cell, the register cell may have its next state dependent on its present

state and inputs or on its inputs only. If the dependency is only on inputs, then cell design for the iterative combinational circuit and attachment of the iterative circuit to flip-flops is appropriate. If, however, the state of the flip-flop is fed back to the inputs of the iterative circuit cell, sequential design methods can also be applied. The next example illustrates simple register cell design in such a case.

EXAMPLE 7-1 Register Cell Design

A register A is to implement the following register transfers:

AND: $A \leftarrow A \wedge B$

EXOR: $A \leftarrow A \oplus B$

OR: $A \leftarrow A \vee B$

Unless specified otherwise, we assume that

1. Only one of AND, EXOR, and OR is equal to 1, and
2. For all of AND, EXOR, and OR equal to 0, the content of A remains unchanged.

A simple design approach for a register cell with conditions 1 and 2 uses a register with parallel load constructed from D flip-flops with Enable (Enable = Load) from Figure 7-2. For this approach, the expression for Load is the OR of all control signals that cause a transfer to occur. The expression for D_i consists of an OR of the AND of each control signal with the operation on the right-hand side of the corresponding transition.

For this example, the resulting equations for LOAD and D_i are

$$\text{LOAD} = \text{AND} + \text{EXOR} + \text{OR}$$

$$D_i = A(t+1)_i = \text{AND} \cdot A_i B_i + \text{EXOR} \cdot (A_i \bar{B}_i + \bar{A}_i B_i) + \text{OR} \cdot (A_i + B_i)$$

The equation for D_i has an implementation similar to that used for the selection part of a multiplexer in which a set of ENABLE blocks drive an OR gate. AND, EXOR, and OR are enabling signals, and the remaining part of the respective terms in D_i consists of the function enabled.

Using D flip-flops for the register storage and no clock gating, a multiplexer must also be implemented in each cell:

$$D_{LFF} = \text{LOAD} \cdot D_i + \overline{\text{LOAD}} \cdot A_i$$

This equation is given to show the hidden cost inside of the basic parallel load register cell.

A more complex approach is to design directly for D flip-flops using a sequential circuit design approach rather than the ad hoc approach based on parallel load flip-flops.

We can formulate a coded state table with A as the state variable and output, and AND, EXOR, OR, and B as inputs, as shown in Table 7-11.

By formulating the flip-flop input equation for $D_i = A(t+1)_i$,

$$D_i = A(t+1)_i = \text{AND} \cdot A_i B_i + \text{EXOR} \cdot (A_i \bar{B}_i + \bar{A}_i B_i) + \text{OR} \cdot (A_i + B_i) \\ + \overline{\text{AND}} \cdot \overline{\text{EXOR}} \cdot \overline{\text{OR}} \cdot A_i$$

□ **TABLE 7-11**
State Table and Flip-Flop Inputs for Counter

Present State A	Next State A(t + 1)						
	(AND = 0)	(OR = 1)	(OR = 1)	(EXOR = 1)	(EXOR = 1)	(AND = 1)	(AND = 1)
	(EXOR=0) (OR=0)	(B=0)	(B=1)	(B=0)	(B=1)	(B=0)	(B=1)
0	0	0	1	0	1	0	0
1	1	1	1	1	0	0	1

Due to the relationship between the OR operator and the AND and EXOR operators and other algebraic reductions, this can be simplified to

$$A(t+1)_i = (OR + AND) \cdot A_i \bar{B}_i + (OR + EXOR) \cdot (A_i \bar{B}_i + \bar{A}_i B_i) + \overline{AND + EXOR} \cdot A_i$$

The terms $OR + AND$, $OR + EXOR$, and $\overline{AND + EXOR}$ do not depend on the values A_i and B_i associated with any of the cells. The logic for these terms can be shared by all of the register cells. Using C_1 , C_2 , and C_3 as intermediate variables, the following set of equations results:

$$C_1 = OR + AND$$

$$C_2 = OR + EXOR$$

$$C_3 = \overline{AND + EXOR}$$

$$D_i = A(t+1)_i = C_1 A_i \bar{B}_i + C_2 (A_i \bar{B}_i + \bar{A}_i B_i) + C_3 \cdot A_i$$

An implementation for register cell A_i and the logic shared by all of the cells is given in Figure 7-17, with the implementation for logic shared by the register cells in A. Before comparing these results with those from the simple approach, we can apply similar simplification and logic sharing to the results of the simple approach:

$$C_1 = OR + AND$$

$$C_2 = OR + EXOR$$

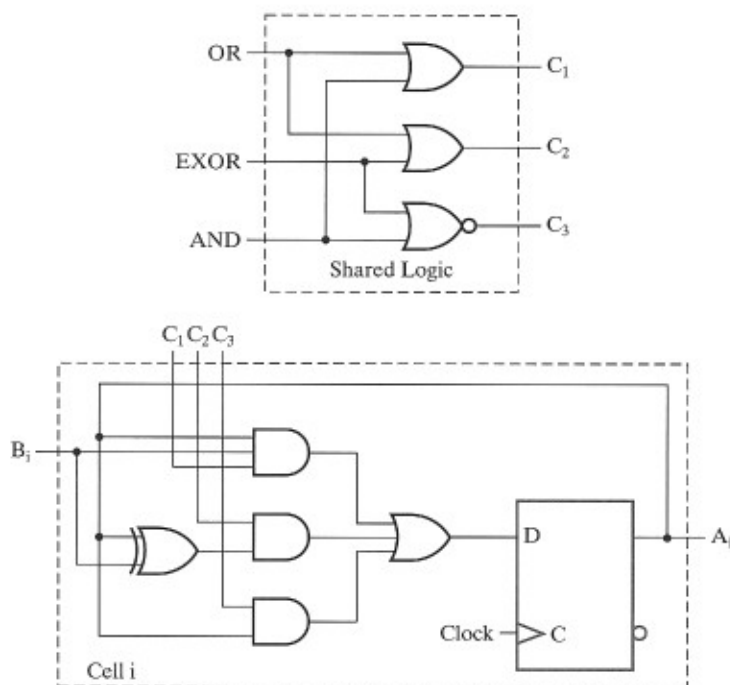
$$D_i = A(t+1)_i = C_1 A_i \bar{B}_i + C_2 (A_i \bar{B}_i + \bar{A}_i B_i)$$

$$LOAD = C_1 + C_2$$

$$D_{i,FF} = LOAD \cdot D_i + \overline{LOAD} \cdot A_i$$

If these equations are used directly the cost of the simple approach is somewhat higher. However, if these equations are provided to a minimization tool instead of being used directly, the same equations as the more complex method will result. Thus, the ease of using the simplest approach does not necessarily cause any increase in hardware cost. ■

In the preceding example, there are no lateral connections between adjacent cells. Among the operations requiring lateral connections are shifts, arithmetic operations, and comparisons. One approach to the design of these structures is to combine combinational designs given in Chapter 5 with selection logic and flip-flops. A generic approach for multifunctional registers using flip-flops with parallel load is shown in Figure 7-8. This simple approach bypasses register cell design, but



□ **FIGURE 7-17**
Logic Diagram - Register Cell Design Example 7-1

if directly implemented, can result in excessive logic and too many lateral connections. The alternative is to do a custom register cell design. In such designs, a critical factor is the definition of the lateral connection(s) needed. Also, different operations can be defined by controlling input to the least significant cell of the cell cascade. The custom design approach is illustrated in the next example by the design of a multifunctional register cell.

EXAMPLE 7-2 Register Cell Design

A register A is to implement the following register transfers:

SHL: $A \leftarrow s1 A$

EXOR: $A \leftarrow A \oplus B$

ADD: $A \leftarrow A + B$

Unless specified otherwise, we assume that

1. Only one of SHL, EXOR, and ADD is equal to 1, and
2. For all of SHL, EXOR, and ADD equal to 0, the content of A remains unchanged.

A simple approach to designing a register cell with conditions 1 and 2 is to use a register with parallel load controlled by LOAD. For this approach, the expression for LOAD is the OR of all control signals that cause a transfer to occur.

The implementation for D_i consists of an AND-OR, with each AND having a control signal and the logic for the operation on the right-hand side as its inputs.

For this example, the resulting equations for LOAD and D_i are

$$\text{LOAD} = \text{SHL} + \text{EXOR} + \text{ADD}$$

$$D_i = A(t+1)_i = \text{SHL} \cdot A_{i-1} + \text{EXOR} \cdot (A_i \oplus B_i) + \text{ADD} \cdot ((A_i \oplus B_i) \oplus C_i)$$

$$C_{i+1} = (A_i \oplus B_i)C_i + A_i B_i$$

These equations can be used without modification or can be optimized.

Now, suppose, instead, that we do a custom design assuming that all of the register cells are identical. This means that the least and most significant cells will be the same as those internal to the cell chain. Because of this, the value of C_0 must be specified and the use, if any, of C_n must be determined for each of the three operations. For the left shift, a zero fill of the vacated rightmost bit is assumed, giving $C_0 = 0$. Since C_0 is not involved in the EXOR operation, it can be assumed to be a don't-care. Finally, for the addition, C_0 either can be assumed to be 0 or can be left as a variable to permit a carry from a previous addition to be injected. We assume that C_0 equals 0 for addition, since no additional carry-in is specified by the register transfer statement.

Our first formulation goal is to minimize lateral connections between cells. Two of the three operations, left shift and addition, require a lateral connection to the left (i.e., toward the most significant end of the cell chain). Our goal is to use one signal for both operations, say, C_i . It already exists for the addition but must be redefined to handle both the addition and the left shift. Also in our custom design, the parallel load flip-flop will be replaced by a D flip-flop. We can now formulate the state table for the register cell shown in Table 7-12:

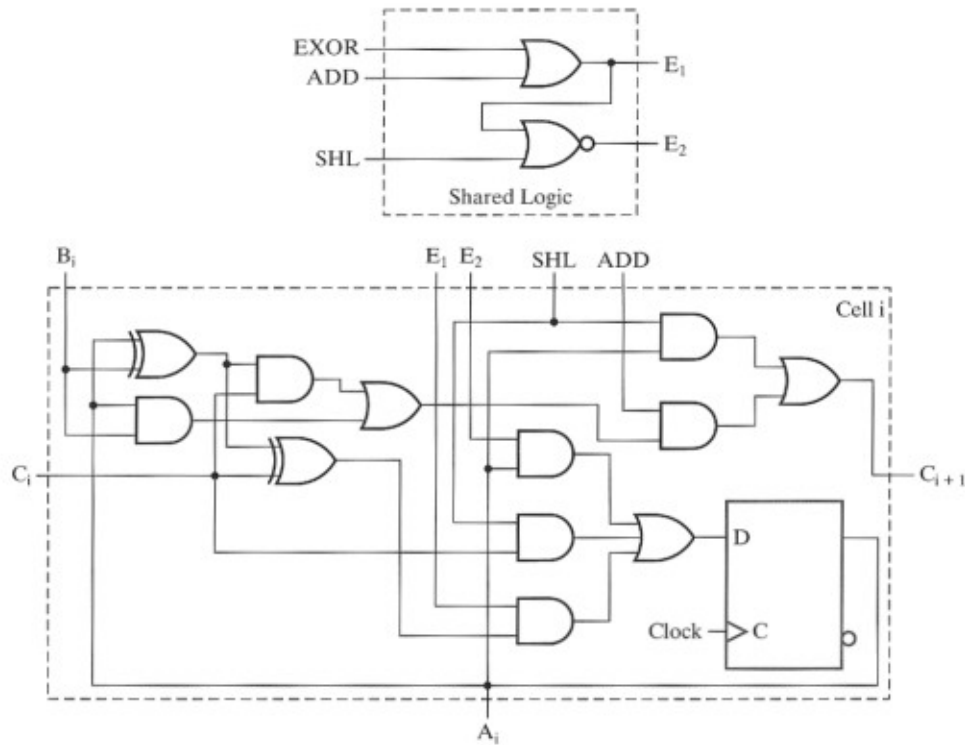
$$D_i = A(t+1)_i = \overline{\text{SHL}} \cdot \overline{\text{EXOR}} \cdot \overline{\text{ADD}} \cdot A_i + \text{SHL} \cdot C_i + \text{EXOR} \cdot (A_i \oplus B_i) + \text{ADD} \cdot (A_i \oplus B_i \oplus C_i)$$

$$C_{i+1} = \text{SHL} \cdot A_i + \text{ADD} \cdot ((A_i \oplus B_i)C_i + A_i B_i)$$

The term $A_i \oplus B_i$ appears in both the EXOR and ADD terms. In fact, if $C_i = 0$ during the EXOR operation, then the functions for the sum in ADD and for EXOR can be identical. In the C_{i+1} equation, since SHL and ADD are both 0 when EXOR is 1, C_i is 0 for all cells in the cascade except the least significant one. For the least

□ TABLE 7-12
State Table and Flip-Flop Inputs for Register Cell Design in Example 7-2

Present State A_i	Inputs		Next State $A_i(t+1)$ /Output C_{i+1}									
	SHL = 0	SHL = 1	1	1	1	EXOR = 1	1	ADD = 1	1	1	1	1
	EXOR = 0	$B_i = 0$	0	1	1	$B_i = 0$	1	$B_i = 0$	0	1	1	1
	ADD = 0	$C_i = 0$	1	0	1			$C_i = 0$	1	0	1	1
0	0/X	0/0	0/0	1/0	0/0	1/0	0/X	1/X	0/0	1/0	1/0	0/1
1	1/X	0/1	0/1	1/1	0/1	1/1	1/X	0/X	1/0	0/1	0/1	1/1



□ **FIGURE 7-18**
Logic Diagram - Register Cell Design Example 7-2

significant cell, the specification states that $C_0 = 0$. Thus, input values C_i are 0 for all cells in register A . So we can combine the ADD and EXOR operations as follows:

$$D_i = A(t+1)_i = \overline{\text{SHL}} \cdot \overline{\text{EXOR}} \cdot \overline{\text{ADD}} \cdot A_i + \text{SHL} \cdot C_i + (\text{EXOR} + \text{ADD}) \cdot ((A_i \oplus B_i) \oplus C_i)$$

The expressions $\overline{\text{SHL}} \cdot \overline{\text{EXOR}} \cdot \overline{\text{ADD}}$ and $\text{EXOR} + \text{ADD}$, that are independent of A_i , B_i , and C_i , can be shared by all cells. The resulting equations are

$$E_1 = \text{EXOR} + \text{ADD}$$

$$E_2 = \overline{E_1} + \text{SHL}$$

$$D_i = E_2 \cdot A_i + \text{SHL} \cdot C_i + E_1 \cdot ((A_i \oplus B_i) \oplus C_i)$$

$$C_{i+1} = \text{SHL} \cdot A_i + \text{ADD} \cdot ((A_i \oplus B_i) C_i + A_i B_i)$$

The resulting register cell appears in Figure 7-18. Comparing this result with the register cell for the simple design, we note the following two differences:

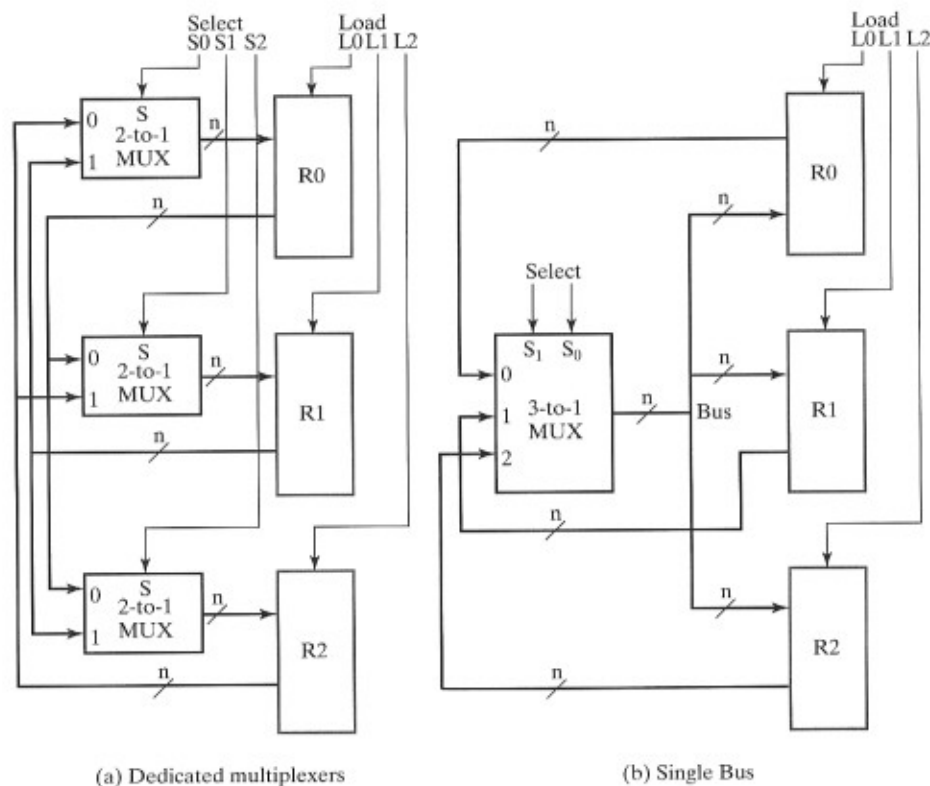
1. Only one lateral connection between cells exists instead of two.
2. Logic has been very efficiently shared by the addition and the EXOR operation.

The custom cell design has produced connection and logic savings not present in the block level design with or without optimization. ■

7-8 MULTIPLEXER AND BUS-BASED TRANSFERS FOR MULTIPLE REGISTERS

A typical digital system has many registers. Paths must be provided to transfer data from one register to another. The amount of logic and the number of interconnections may be excessive if each register has its own dedicated set of multiplexers. A more efficient scheme for transferring data between registers is a system that uses a shared transfer path called a *bus*. A bus is characterized by a set of common lines, with each line driven by selection logic. Control signals for the logic select a single source and one or more destinations on any clock cycle for which a transfer occurs.

In Section 7-4, we saw that multiplexers and parallel load registers can be used to implement dedicated transfers from multiple sources. A block diagram for such transfers between three registers is shown in Figure 7-19(a). There are three n -bit 2-to-1 multiplexers, each with its own select signal. Each register has its own load signal. The same system based on a bus can be implemented by using a single n -bit 3-to-1 multiplexer and parallel load registers. If a set of multiplexer outputs is shared as a common path, these output lines are a bus. Such a system with a single



□ **FIGURE 7-19**
Single Bus versus Dedicated Multiplexers

□ TABLE 7-13

Examples of Register Transfers Using the Single Bus
in Figure 7-19(b)

Register Transfer	Select		Load		
	S1	S0	L2	L1	L0
$R0 \leftarrow R2$	1	0	0	0	1
$R0 \leftarrow R1, R2 \leftarrow R1$	0	1	1	0	1
$R0 \leftarrow R1, R1 \leftarrow R0$	Impossible				

bus for transfers between three registers is shown in Figure 7-19(b). The control input pair, Select, determines the contents of the single source register that will appear on the multiplexer outputs (i.e., on the bus). The load inputs determine the destination register or registers to be loaded with the bus data.

In Table 7-13, transfers using the single-bus implementation of Figure 7-19(b) are illustrated. The first transfer is from $R2$ to $R0$. Select equals 10, selecting input $R2$ to the multiplexer. Load signal $L0$ for register $R0$ is 1, with all other loads at 0, causing the contents of $R2$ on the bus to be loaded into $R0$ on the next positive clock transition. The second transfer in the table illustrates the loading of the contents of $R1$ into both $R0$ and $R2$. The source $R1$ is selected because Select is equal to 01. In this case, $L2$ and $L0$ are both 1, causing the contents of $R1$ on the bus to be loaded into registers $R0$ and $R2$. The third transfer, an exchange between $R0$ and $R1$, is impossible in a single clock cycle, since it requires two simultaneous sources, $R0$ and $R1$, on the single bus. Thus, this transfer requires at least two buses or a bus combined with a dedicated path from one of the registers to the other. Note that such a transfer can be executed on the dedicated multiplexers in Figure 7-19(a). So, for a single-bus system, simultaneous transfers with different sources in a single clock cycle are impossible, whereas for the dedicated multiplexers, any combination of transfers is possible. Hence, the reduction in hardware that occurs for a single bus in place of dedicated multiplexers results in limitations on simultaneous transfers.

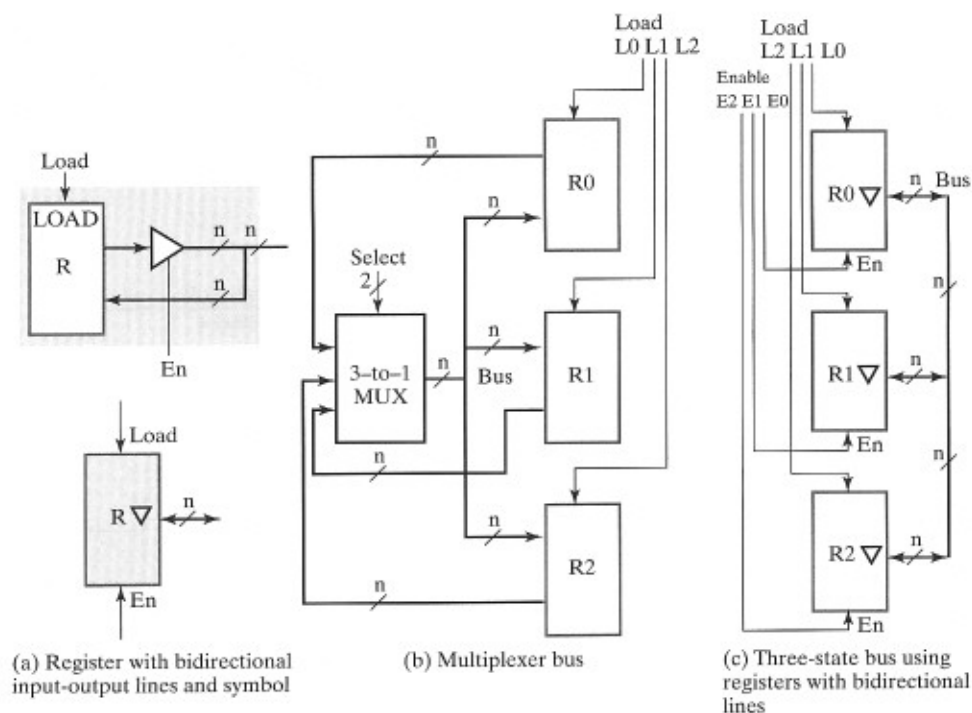
If we assume that only single-source transfers are needed, then we can use Figure 7-19 to compare the complexity of the hardware in dedicated versus bus-based systems. First of all, assume a multiplexer design, as in Figure 4-16. In Figure 7-19(a), there are $2n$ AND gates and n OR gates per multiplexer (not counting inverters), for a total of $9n$ gates. In contrast, in Figure 7-19(b), the bus multiplexer requires only $3n$ AND gates and n OR gates, for a total of $4n$ gates. Also, the data input connections to the multiplexers are reduced from $6n$ to $3n$. Thus, the cost of the selection hardware is reduced by about half.

Three-State Bus

A bus can be constructed with the three-state buffers introduced in Section 2-8 instead of multiplexers. This has the potential for additional reductions in the number of connections. But why use three-state buffers instead of a multiplexer, particularly for implementing buses? The reason is that many three-state buffer outputs

can be connected together to form a bit line of a bus, and this bus is implemented using only one level of logic gates. On the other hand, in a multiplexer, such a large number of sources means a high fan-in OR, which requires multiple levels of OR gates, introducing more logic and increasing delay. In contrast, three-state buffers provide a practical way to construct fast buses with many sources, so they are often preferred in such cases. More important, however, is the fact that signals can travel in two directions on a three-state bus. Thus, the three-state bus can use the same interconnection to carry signals into and out of a logic circuit. This feature, which is most important when crossing chip boundaries, is illustrated in Figure 7-20(a). The figure shows a register with n lines that serve as both inputs and outputs lying across the boundary of the shaded area. If the three-state buffers are enabled, then the lines are outputs; if the three-state buffers are disabled, then the lines can be inputs. The symbol for this structure is also given in the figure. Note that the bidirectional bus lines are represented by a two-headed arrow. Also, a small inverted triangle denotes the three-state outputs of the register.

Figure 7-20(b) and Figure 7-20(c) show a multiplexer-implemented bus and a three-state bus, respectively, for comparison. The symbol from Figure 7-20(a) for a register with bidirectional input-output lines is used in Figure 7-20(c). In contrast to the situation in Figure 7-19, where dedicated multiplexers were replaced by a bus, these two implementations are identical in terms of their register transfer



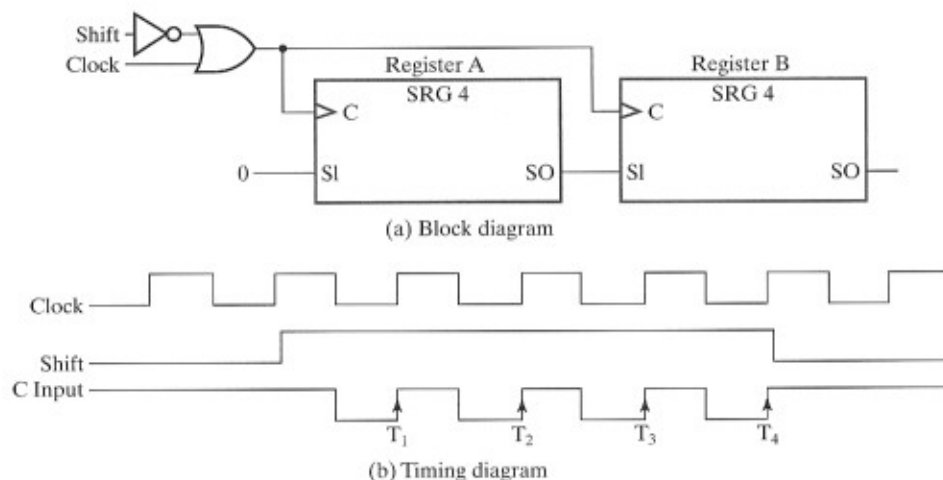
□ **FIGURE 7-20**
Three-State Bus versus Multiplexer Bus

capability. Note that, in the three-state bus, there are only three data connections to the set of register blocks for each bit of the bus. The multiplexer-implemented bus has six data connections per bit to the set of register blocks. This reduction in the number of data connections by half, along with the ability to easily construct a bus with many sources, makes the three-state bus an attractive alternative. The use of such bidirectional input-output lines is particularly effective between logic circuits in different physical packages.

7-9 SERIAL TRANSFER AND MICROOPERATIONS

A digital system is said to operate in a serial mode when information in the system is transferred or manipulated one bit at a time. Information is transferred one bit at a time by shifting the bits out of one register and into a second register. This transfer method is in contrast to parallel transfer, in which all the bits of the register are transferred at the same time.

The serial transfer of information from register *A* to register *B* is done with shift registers, as shown in the block diagram of Figure 7-21(a). The serial output of register *A* is connected to the serial input of register *B*. The serial input of register *A* receives 0's while its data are transferred to register *B*. It is also possible for register *A* to receive other binary information, or if we want to maintain the data in register *A*, we can connect its serial output to its serial input so that the information is circulated back into the register. The initial content of register *B* is shifted out through its serial output and is lost unless it is transferred back into register *A*, to a third shift register, or to other storage. The shift control input *Shift* determines when and how many times the registers are shifted. The registers using *Shift* are controlled by means of the logic from Figure 7-2, which allows the clock pulses to pass to the shift register clock inputs only when *Shift* has the value logic 1.



□ **FIGURE 7-21**
Serial Transfer

In Figure 7-21, each shift register has four stages. The logic that supervises the transfer must be designed to enable the shift registers, through the *Shift* signal, for a fixed time of four clock pulses. Shift register enabling is shown in the timing diagram for the clock gating logic in Figure 7-21(b). Four pulses find *Shift* in the active state, so that the output of the logic connected to the clock inputs of the registers produces four pulses: T_1 , T_2 , T_3 , and T_4 . Each positive transition of these pulses causes a shift in both registers. After the fourth pulse, *Shift* changes back to 0 and the shift registers are disabled. We note again that, for positive-edge triggering, the pulses on the clock inputs are 0, and the inactive level when no pulses are present is a 1 rather than a 0.

Now suppose that the binary content of register *A* before the shift is 1011, that of register *B* is 0010, and the *SI* of register *A* is logic 0. Then the serial transfer from *A* to *B* occurs in four steps, as shown in Table 7-14. With the first pulse T_1 , the rightmost bit of *A* is shifted into the leftmost bit of *B*, the leftmost bit of *A* receives a 0 from the serial input, and at the same time, all other bits of *A* and *B* are shifted one position to the right. The next three pulses perform identical operations, shifting the bits of *A* into *B* one at a time while transferring 0's to *A*. After the fourth shift, the logic supervising the transfer changes the *Shift* signal to 0 and the shifts stop. Register *B* contains 1011, which is the previous value of *A*. Register *A* contains all 0's.

The difference between serial and parallel modes of operation should be apparent from this example. In the parallel mode, information is available from all bits of a register, and all bits can be transferred simultaneously during one clock pulse. In the serial mode, the registers have a single serial input and a single serial output, and information is transferred one bit at a time.

Serial Addition

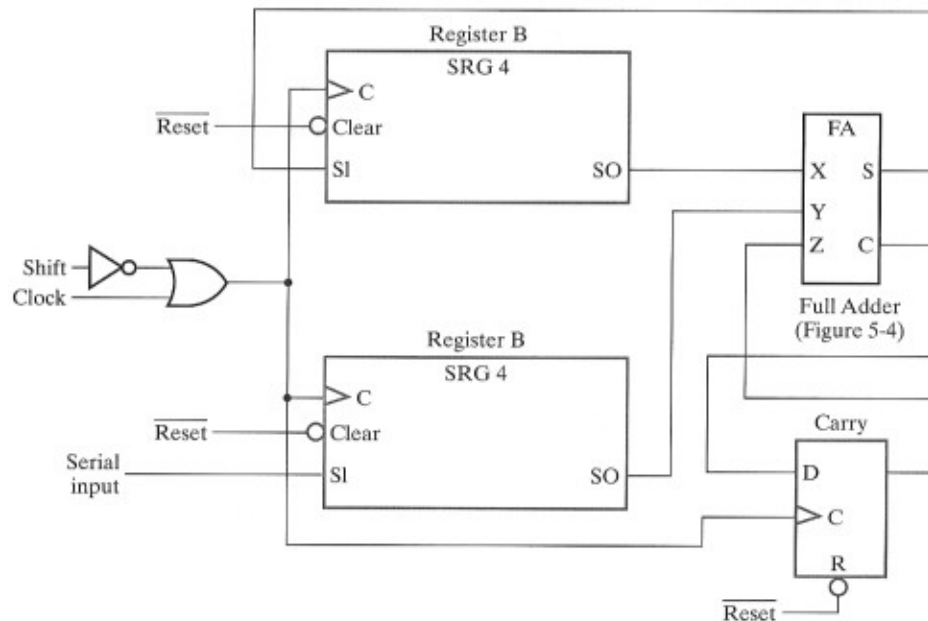
Operations in digital computers are usually done in parallel because of the faster speed attainable. Serial operations are slower, but have the advantage of requiring less hardware. To demonstrate the serial mode of operation, we will show the operation of a serial adder. Also, we compare the serial adder to the parallel counterpart presented in Section 5-2 to illustrate the time-space trade-off in design.

□ **TABLE 7-14**
Example of Serial Transfer

Timing pulse	Shift Register A				Shift Register B			
Initial value	1	0	1	1	0	0	1	0
After T_1	0	1	0	1	1	0	0	1
After T_2	0	0	1	0	1	1	0	0
After T_3	0	0	0	1	0	1	1	0
After T_4	0	0	0	0	1	0	1	1

The two binary numbers to be added serially are stored in two shift registers. Bits are added, one pair at a time, through a single full-adder (FA) circuit, as shown in Figure 7-22. The carry out of the full adder is transferred into a *D* flip-flop. The output of this carry flip-flop is then used as the carry input for the next pair of significant bits. The sum bit on the *S* output of the full adder could be transferred into a third shift register, but we have chosen to transfer the sum bits into register *A* as the contents of the register are shifted out. The serial input of register *B* can receive a new binary number as its contents are shifted out during the addition.

The operation of the serial adder is as follows: Register *A* holds the augend, register *B* holds the addend, and the carry flip-flop has been reset to 0. The serial outputs of *A* and *B* provide a pair of significant bits for the full adder at *X* and *Y*. The output of the carry flip-flop provides the carry input at *Z*. When *Shift* is set to 1, the OR gate enables the clock for both registers and the flip-flop. Each clock pulse shifts both registers once to the right, transfers the sum bit from *S* into the leftmost flip-flop of *A*, and transfers the carry output into the carry flip-flop. Shift control logic enables the registers for as many clock pulses as there are bits in the registers (four pulses in this example). For each pulse, a new sum bit is transferred to *A*, a new carry is transferred to the flip-flop, and both registers are shifted once to the right. This process continues until the shift control logic changes *Shift* to 0. Thus, the addition is accomplished by passing each pair of bits and the previous carry through a single full-adder circuit and transferring the sum, one bit at a time, back into register *A*.



□ **FIGURE 7-22**
Serial Addition

Initially, we can reset register *A*, register *B*, and the *Carry* flip-flop to 0. Then we shift the first number into *B*. Next, the first number from *B* is added to the 0 in *A*. While *B* is being shifted through the full adder, we can transfer a second number to it through its serial input. The second number can be added to the contents of register *A* at the same time that a third number is transferred serially into register *B*. Serial addition may be repeated to form the addition of two, three, or more numbers, with their sum accumulated in register *A*.

A comparison of the serial adder with the parallel adder described in Section 5-2 provides an example of space-time trade-off. The parallel adder has *n* full adders for *n*-bit operands, whereas the serial adder requires only one full adder. Excluding the registers from both, the parallel adder is a combinational circuit, whereas the serial adder is a sequential circuit because it includes the carry flip-flop. The serial circuit also takes *n* clock cycles to complete an addition. Identical circuits, such as the *n* full adders in the parallel adder, connected together in a chain constitute an example of an *iterative logic array*. If the values on the carries between the full adders are regarded as state variables, then the states from the least significant end to the most significant end are the same as the states appearing in sequence on the flip-flop output in the serial adder. Note that in the iterative logic array the states appear in space, but in the sequential circuit the states appear in time. By converting from one of these implementations to the other, one can make a space-time trade-off. The parallel adder in space is *n* times larger than the serial adder (ignoring the area of the carry flip-flop), but it is *n* times faster. The serial adder, although it is *n* times slower, is *n* times smaller in space. This gives the designer a significant choice in emphasizing speed or area, where more area translates into more cost.

7-10 HDL REPRESENTATION FOR SHIFT REGISTERS AND COUNTERS—VHDL

Examples of shift register and a binary counter illustrate the use of VHDL in representing registers and operations on register content.

EXAMPLE 7-3 VHDL for a 4-Bit Shift Register

The VHDL code in Figure 7-23 describes a 4-bit left shift register at the behavioral level. A **RESET** input is present that directly resets the register contents to zero. The shift register contains flip-flops and so has a process description resembling that of a *D* flip-flop. The four flip-flops are represented by the signal **shift**, of type `std_logic_vector` of size four. **Q** cannot be used to represent the flip-flops since it is an output and the flip-flop outputs must be used internally. The left shift is achieved by applying the concatenation operator **&** to the right three bits of **shift** and to shift input **SI**. This quantity is transferred to **shift** moving the contents one bit to the left and loading the value of **SI** into the rightmost bit. Following the process that performs the shift are two statements, one which assigns the

```

-- 4-bit Shift Register with Reset

library ieee;
use ieee.std_logic_1164.all;

entity srg_4_r is
    port(CLK, RESET, SI : in std_logic;
          Q : out std_logic_vector(3 downto 0);
          SO : out std_logic);
end srg_4_r;

architecture behavioral of srg_4_r is
    signal shift : std_logic_vector(3 downto 0);
begin
    process (RESET, CLK)
    begin
        if (RESET = '1') then
            shift <= "0000";
        elsif (CLK'event and (CLK = '1')) then
            shift <= shift(2 downto 0) & SI;
        end if;
    end process;
    Q <= shift;
    SO <= shift(3);
end behavioral;

```

□ FIGURE 7-23

Behavioral VHDL Description of 4-bit Left Shift Register with Direct Reset

value in *shift* to output *Q* and the other which defines the shift out signal *SO* as the contents of the leftmost bit of *shift*. ■

EXAMPLE 7-4 VHDL for a 4-Bit Counter

The VHDL code in Figure 7-24 describes a 4-bit counter at the behavioral level. A *RESET* input is present that directly resets the counter contents to zero. The counter contains flip-flops and, therefore, has a process description resembling that of a *D* flip-flop. The four flip-flops are represented by the signal *count*, of type *std_logic_vector* and of size four. *Q* cannot be used to represent the flip-flops since it is an output and the flip-flop outputs must be used internally. Counting up is achieved by adding 1 in the form of "0001" to *count*. Since addition is not a normal operation on type *std_logic_vector*, it is necessary to use an additional package from the *ieee* library, *std_logic_unsigned.all*, which defines unsigned number operations on type *std_logic*. Following the process that performs reset and counting are two statements, one which assigns the value in *count* to output *Q* and the other which defines the count out signal *CO*. A **when-else** statement is used in which *CO* is set to 1 only for the maximum count with *EN* equal to 1. ■

```

-- 4-bit Binary Counter with Reset

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity count_4_r is
    port(CLK, RESET, EN : in std_logic;
          Q : out std_logic_vector(3 downto 0);
          CO : out std_logic);
end count_4_r;

architecture behavioral of count_4_r is
    signal count : std_logic_vector(3 downto 0);
begin
    process (RESET, CLK)
    begin
        if (RESET = '1') then
            count <= "0000";
        elsif (CLK'event and (CLK = '1') and (EN = '1')) then
            count <= count + "0001";
        end if;
    end process;
    Q <= count;
    CO <= '1' when count = "1111" and EN = '1' else '0';
end behavioral;

```


□ **FIGURE 7-24**
Behavioral VHDL Description of 4-bit Binary Counter with Direct Reset

7-11 HDL REPRESENTATION FOR SHIFT REGISTERS AND COUNTERS—VERILOG

Examples of shift register and binary counter illustrate the use of Verilog in representing registers and operations on register content.

EXAMPLE 7-5 Verilog Code for a Shift Register

The Verilog description in Figure 7-25 describes a left shift register at the behavioral level. A RESET input is present that directly resets the register contents to zero. The shift register contains flip-flops, so has a process description beginning with **always** resembling that of a *D* flip-flop. The four flip-flops are represented by the vector *Q*, of type **reg** with bits numbered 3 down to 0. The left shift is achieved by applying { } to concatenate the right three bits of *Q* and shift input *SI*. This quantity is transferred to *Q*, moving the contents one bit to the left and loading the value of *SI* into the rightmost bit. Just prior to the process that performs the shift



```
// 4-bit Left Shift Register with Reset

module srg_4_r_v (CLK, RESET, SI, Q, SO);
    input CLK, RESET, SI;
    output [3:0] Q;
    output SO;

    reg [3:0] Q;

    assign SO = Q[3];

    always@(posedge CLK or posedge RESET)
    begin
        if (RESET)
            Q <= 4'b0000;
        else
            Q <= {Q[2:0], SI};
    end
endmodule
```

□ FIGURE 7-25

Behavioral Verilog Description of 4-bit Left Shift Register with Direct Reset

is a continuous assignment statement that assigns the contents of the leftmost bit of *Q* to the shift output signal *SO*. ■

EXAMPLE 7-6 Verilog Code for a Counter

The Verilog description in Figure 7-26 describes a 4-bit binary counter at the behavioral level. A *RESET* input is present that directly resets the register contents to zero. The counter contains flip-flops and, therefore, the description contains a process resembling that for a *D* flip-flop. The four flip-flops are represented by the signal *Q* of type **reg** and size four. Counting up is achieved by adding 1 to *Q*. Prior to the process that performs reset and counting is a conditional continuous assignment statement that defines the count out signal *CO*. *CO* is set to 1 only for the maximum count and *EN* equal to 1. Note that logical AND is denoted by **&&**. ■

7-12 CHAPTER SUMMARY

Registers are sets of flip-flops, or interconnected sets of flip-flops, and combinational logic. The simplest registers are flip-flops that are loaded with new contents from their inputs on every clock cycle. More complex are registers in which the flip-flops can be loaded with new contents under the control of a signal on only selected clock cycles. Register transfers are a means of representing and specifying elementary processing operations. Register transfers can be related to corresponding digital system hardware, both at the block diagram


```
// 4-bit Binary Counter with Reset

module count_4_r_v (CLK, RESET, EN, Q, CO);
    input CLK, RESET, EN;
    output [3:0] Q;
    output CO;

    reg [3:0] Q;

    assign CO = (count == 4'b1111 && EN == 1'b1) ? 1 : 0;
    always@(posedge CLK or posedge RESET)
        begin
            if (RESET)
                Q <= 4'b0000;
            else if (EN)
                Q <= Q + 4'b0001;
            end
        endmodule
```

□ FIGURE 7-26

Behavioral Verilog Description of 4-bit Binary Counter with Direct Reset

level and at the detailed logic level. Microoperations are elementary operations performed on data stored in registers. Arithmetic microoperations include addition and subtraction, which are described as register transfers and are implemented with corresponding hardware. Logic microoperations—that is, the bitwise application of logic primitives such as AND, OR, and XOR, combined with a binary word—provide masking and selective complementing on other binary words. Left- and right-shift microoperations move data laterally one or more bit positions at a time.

Shift registers add a new dimension to data transfer, since they are designed to move information laterally one or more bit positions at a time. When combined with the ability to be loaded with data, a shift register can be used to convert data presented in parallel into data presented serially. Likewise, if the outputs of the register are accessible, a shift register can be used to convert data presented serially into data presented in parallel. This lateral movement of data can also be used in hardware structures that perform serial arithmetic operations.

Counters are used to provide a sequence of values, often in binary counting order. The simplest of counters has no inputs other than an asynchronous reset for initialization to zero. This kind of counter simply counts clock pulses. More complex versions can also be loaded with data and have input signals that enable them to count.

Multiplexers select among multiple transfer paths entering a register. Buses are shared register transfer paths for multiple registers and offer reduced hardware in trade for limitations on possible simultaneous transfers. In addition to multiplexers, three-state buffers enhance the implementation of buses by providing bidirectional transfer paths and reduced connections.

REFERENCES

1. MANO, M. M. *Digital Design*, 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 2002.
2. WAKERLY, J. F. *Digital Design: Principles and Practices*, 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2000.
3. *IEEE Standard VHDL Language Reference Manual*. (ANSI/IEEE Std 1076-1993; revision of IEEE Std 1076-1987). New York: The Institute of Electrical and Electronics Engineers, 1994.
4. *IEEE Standard Description Language Based on the Verilog(TM) Hardware Description Language* (IEEE Std 1364-1995). New York: The Institute of Electrical and Electronics Engineers, 1995.
5. THOMAS, D. E., AND P. R. MOORBY. *The Verilog Hardware Description Language* 4th ed. Boston: Kluwer Academic Publishers, 1998.

PROBLEMS



The plus (+) indicates a more advanced problem and the asterisk (*) indicates a solution is available on the Companion Website for the text.

- 7-1. Use manual or computer-based simulation to demonstrate that the clock gating function in Figure 7-1(c) works properly with the register in Figure 7-1(a). Use a positive-edge-triggered flip-flop with *Clock* as its clock input to generate *Load*. Be sure to use nonzero gate and flip-flop delays.
- 7-2. +Change the OR gate in Figure 7-1(c) into an AND gate, and remove the inverter on *Load*.
 - (a) Perform the same simulation as in Problem 7-1 to demonstrate that the new clock gating circuitry does not work correctly. Explain what goes wrong.
 - (b) Will the circuit work correctly if the flip-flop generating *Load* is triggered by the negative rather than the positive edge of *Clock*?
- 7-3. Assume that registers *R1* and *R2* in Figure 7-6 hold two unsigned numbers. When select input *X* is equal to 1, the adder-subtractor circuit performs the arithmetic operation " $R1 + 2$'s complement of $R2$." This sum and the output carry C_n are transferred into *R1* and *C* when $K_1 = 1$ and a positive edge occurs on the clock.
 - (a) Show that if $C = 1$, then the value transferred to *R1* is equal to $R1 - R2$, but if $C = 0$, the value transferred to *R1* is the 2's complement of $R2 - R1$.
 - (b) Indicate how the value in the *C* bit can be used to detect a borrow after the subtraction of two unsigned numbers.
- 7-4. *Perform the bitwise logic AND, OR, and XOR of the two 8-bit operands 10011001 and 11000011.

- 7-5. Given the 16-bit operand 00001111 10101010, what operation must be performed and what operand must be used
- (a) to clear all even bit positions to 0? (Assume bit positions are 15 through 0 from left to right.)
 - (b) to set the leftmost 4 bits to 1?
 - (c) to complement the center 8 bits?
- 7-6. *Starting from the 8-bit operand 01010011, show the values obtained after each shift microoperation given in Table 7-5.
- 7-7. *Modify the register of Figure 7-11 so that it will operate according to the following function table using mode selection inputs S_1 and S_0 .

S_1	S_0	Register Operation
0	0	No change
0	1	Clear register to 0
1	0	Shift down
1	1	Load parallel data

- 7-8. *A ring counter is a shift register, as in Figure 7-9, with the serial output connected to the serial input.
- (a) Starting from an initial state of 1000, list the sequence of states of the four flip-flops after each shift.
 - (b) Beginning in state 10...0, how many states are there in the count sequence of an n -bit ring counter?
- 7-9. A switch-tail ring counter (Johnson counter) uses the complement of the serial output of a right shift register as its serial input.
- (a) Starting from an initial state of 0000, list the sequence of states after each shift until the register returns to 0000.
 - (b) Beginning in state 00...0, how many states are there in the count sequence of an n -bit switch-tail counter?
- 7-10. How many flip-flop values are complemented in an 8-bit binary ripple counter to reach the next count value after
- (a) 11101111? (b) 01111111?
- 7-11. + For the CMOS logic family, the power consumption is proportional to the sum of the changes from 1-to-0 and 0-to-1 on all gate inputs and outputs in the circuit. When designing counters in very low power circuits, ripple counters are preferred over regular synchronous binary counters. Carefully count the numbers of changing inputs and outputs, including those related to the clock for a complete cycle of values in a 4-bit ripple counter versus a regular synchronous counter of the same length. Based on this examination, explain why the ripple counter is superior in terms of power consumption.
- 7-12. Construct a 16-bit serial-parallel counter, using four 4-bit parallel counters. Suppose all added logic is AND gates and serial connections are employed

between the four counters. What is the maximum number of AND gates in a chain that a signal must propagate through in the 16-bit counter?

- 7-13.** +A 64-bit synchronous parallel counter is to be designed.
- (a)** Draw the logic diagram of a 64-bit parallel counter, using 8-bit parallel counter blocks and two levels of parallel gating connections between the blocks. In these blocks, *CO* is not driven by *EN*.
 - (b)** What is the ratio of the maximum frequency of operation of this counter to that of a 64-bit serial-parallel counter? Assume that the *D* flip-flop propagation time is twice the delay of an AND gate and that the flip-flop setup time is equal to the delay of an AND gate.
- 7-14.** Using the synchronous binary counter of Figure 7-13 and an AND gate, construct a counter that counts from 0000 through 1010. Repeat for a count from 0000 to 1110. Minimize the number of inputs to the AND gate.
- 7-15.** Using two binary counters of the type shown in Figure 7-13 and logic gates, construct a binary counter that counts from decimal 9 through decimal 129. Add an additional input to the counter that initializes it synchronously to 9 when the signal INIT is 1.
- 7-16.** *Verify the flip-flop input equations of the synchronous BCD counter specified in Table 7-9. Draw the logic diagram of the BCD counter with a count enable input.
- 7-17.** *Use *D* flip-flops and gates to design a binary counter with each of the following repeated binary sequences:
- (a)** 0, 1, 2 **(b)** 0, 1, 2, 3, 4, 5
- 7-18.** Use *D*-type flip-flops and gates to design a counter with the following repeated binary sequence: 0, 1, 3, 2, 4, 6.
- 7-19.** Use only *D*-type flip-flops to design a counter with the following repeated binary sequence: 0, 1, 2, 4, 8.
- 7-20.** Draw the logic diagram of a 4-bit register with mode selection inputs S_1 and S_0 . The register is to be operated according to the following function table:

S_1	S_0	Register Operation
0	0	No change
0	1	Clear register to 0
1	0	Complement output
1	1	Load parallel data

- 7-21.** *Show the diagram of the hardware that implements the register transfer statement

$$C_3: R2 \leftarrow R1, R1 \leftarrow R2$$

- 7-22.** The outputs of registers $R0$, $R1$, $R2$, and $R3$ are connected through 4-to-1 multiplexers to the inputs of a fourth register $R4$. Each register is 8 bits long. The required transfers, as dictated by four control variables, are

$$C_0: R4 \leftarrow R0$$

$$C_1: R4 \leftarrow R1$$

$$C_2: R4 \leftarrow R2$$

$$C_3: R4 \leftarrow R3$$

The control variables are mutually exclusive (i.e., only one variable can be equal to 1 at any time) while the other three are equal to 0. Also, no transfer into $R4$ is to occur for all control variables equal to 0. (a) Using registers and a multiplexer, draw a detailed logic diagram of the hardware that implements a single bit of these register transfers. (b) Draw a logic diagram of the simple logic that maps the control variables as inputs to outputs that are the two select variables for the multiplexers to outputs that are the load signals for registers.

- 7-23.** *Using two 4-bit registers $R1$ and $R2$, and AND gates, OR gates, and inverters, draw one bit slice of the logic diagram that implements all of the following statements:

$$C_0: R2 \leftarrow 0 \quad \text{Clear } R2 \text{ synchronously with the clock}$$

$$C_1: R2 \leftarrow \overline{R2} \quad \text{Complement } R2$$

$$C_2: R2 \leftarrow R1 \quad \text{Transfer } R1 \text{ to } R2$$

The control variables are mutually exclusive (i.e., only one variable can be equal to 1 at any time) while the other two are equal to 0. Also, no transfer into $R2$ is to occur for all control variables equal to 0.

- 7-24.** A register cell is to be designed for an 8-bit register A that has the following register transfer functions:

$$C_0: A \leftarrow A \wedge B$$

$$C_1: A \leftarrow A \vee \overline{B}$$

Find optimum logic using AND, OR, and NOT gates for the D input to the D flip-flop in the cell.

- 7-25.** A register cell is to be designed for an 8-bit register $R0$ that has the following register transfer functions:

$$\overline{S_1} \cdot \overline{S_0}: R0 \leftarrow 0$$

$$\overline{S_1} \cdot S_0: R0 \leftarrow R0 \vee R1$$

$$\begin{aligned} S_1 \cdot \overline{S_0}: R0 &\leftarrow R0 \oplus R1 \\ S_1 \cdot S_0: R0 &\leftarrow R0 \wedge R1 \end{aligned}$$

Find optimum logic using AND, OR, and NOT gates for the D input to the D flip-flop in the cell.

- 7-26. A register cell is to be designed for register *B*, which has the following register transfers:

$$\begin{aligned} S_1: B &\leftarrow B + A \\ S_0: B &\leftarrow B + 1 \end{aligned}$$

Share the combinational logic between the two transfers as much as possible.

- 7-27. Logic to implement transfers among three registers, *R0*, *R1*, and *R2*, is to be implemented. Use the control variable assumptions given in Problem 7-2. The register transfers are as follows:

$$\begin{aligned} C_A: R1 &\leftarrow R0 \\ C_B: R0 &\leftarrow R1, R2 \leftarrow R0 \\ C_C: R1 &\leftarrow R2, R0 \leftarrow R2 \end{aligned}$$

Using registers and dedicated multiplexers, draw a detailed logic diagram of the hardware that implements a single bit of these register transfers.

Draw a logic diagram of simple logic that converts the control variables C_A , C_B , and C_C as inputs to outputs that are the SELECT inputs for the multiplexers and LOAD signals for the registers.

- 7-28. *Two register transfer statements are given (otherwise, *R1* is unchanged):

$$\begin{aligned} C_1: R1 &\leftarrow R1 + R2 && \text{Add } R2 \text{ to } R1 \\ \overline{C_1} C_2: R1 &\leftarrow R1 + 1 && \text{Increment } R1 \end{aligned}$$

- (a) Using a 4-bit counter with parallel load as in Figure 7-13 and a 4-bit adder as in Figure 5-5, draw the logic diagram that implements these register transfers.
(b) Repeat part (a) using a 4-bit adder as in Figure 5-5 plus external gates as needed. Compare with the implementation in part (a).

- 7-29. Repeat Problem 7-27 using one multiplexer-based bus and one direct connection from one register to another instead of dedicated multiplexers.
7-30. Draw a logic diagram of a bus system similar to the one shown in Figure 7-7, but use three-state buffers and a decoder instead of the multiplexers.
7-31. *A system is to have the following set of register transfers, implemented using buses:

$$C_a: R0 \leftarrow R1$$

$$C_b: R3 \leftarrow R1, R1 \leftarrow R4, R4 \leftarrow R0$$

$$C_c: R2 \leftarrow R3, R0 \leftarrow R2$$

$$C_d: R2 \leftarrow R4, R4 \leftarrow R2$$

- (a) For each destination register, list all of the source registers.
 - (b) For each source register, list all of the destination registers.
 - (c) With consideration for which of the transfers must occur simultaneously, what is the minimum number of buses that can be used to implement the set of transfers? Assume that each register will have a single bus as its input.
 - (d) Draw a block diagram of the system, showing the registers and buses and the connections between them.
- 7-32. The following register transfers are to be executed in, at most, two clock cycles:

$$R0 \leftarrow R1$$

$$R2 \leftarrow R1$$

$$R4 \leftarrow R2$$

$$R6 \leftarrow R3$$

$$R8 \leftarrow R3$$

$$R9 \leftarrow R4$$

$$R10 \leftarrow R4$$

$$R11 \leftarrow R1$$

- (a) What is the minimum number of buses required? Assume that only one bus can be attached to a register input and that any net connected to a register input is counted as a bus.
 - (b) Draw a block diagram connecting registers and multiplexers to implement the transfers.
- 7-33. What is the minimum number of clock cycles required to perform the following set of register transfers using two buses?

$$R0 \leftarrow R1$$

$$R7 \leftarrow R1$$

$$R2 \leftarrow R3$$

$$R8 \leftarrow R4$$

$$R5 \leftarrow R6$$

$$R9 \leftarrow R3$$

Assume that only one bus can be attached to a register input and that any net connected to a register input is counted as a bus.

- 7-34. *The content of a 4-bit register is initially 0000. The register is shifted eight times to the right, with the sequence 10110001 as the serial input. The leftmost bit of the sequence is applied first. What is the content of the register after each shift?
- 7-35. *The serial adder of Figure 7-22 uses two 4-bit registers. Register *A* holds the binary number 0111 and register *B* holds 0101. The carry flip-flop is initially

reset to 0. List the binary values in register *A* and the carry flip-flop after each of four shifts.



All files referred to in the remaining problems are available in ASCII form for simulation and editing on the Companion Website for the text. A VHDL or Verilog compiler/simulator is necessary for the problems or portions of problems requesting simulation. Descriptions can still be written, however, for many problems without using compilation or simulation.

- 7-36. *Write a behavioral VHDL description for the 4-bit register in Figure 7-1 (a). Compile and simulate your description to demonstrate correctness.
- 7-37. Repeat problem 7-36 for the 4-bit register with parallel load in Figure 7-2.
- 7-38. Write a VHDL description for the 4-bit binary counter in Figure 7-13 using a register for the *D* flip-flops and Boolean equations for the logic. Compile and simulate your description to demonstrate correctness.
- 7-39. *Write a behavior Verilog description for the 4-bit register in Figure 7-1 (a). Compile and simulate your description to demonstrate correctness.
- 7-40. Repeat problem 5-39 for the 4-bit register with parallel load in Figure 7-2.
- 7-41. Write a Verilog description for the 4-bit binary counter in Figure 7-13 using a register for the *D* flip-flops and Boolean equations for the logic. Compile and simulate your description to demonstrate correctness.

CHAPTER

8

SEQUENCING AND CONTROL

In Chapter 7, we introduced the concept of the datapath for processing data and datapath implementation using registers and register transfers. In the current chapter, our focus is the control unit also introduced in Chapter 7. Digital systems can be classified as programmable or nonprogrammable systems depending on the type of control unit. Nonprogrammable systems have inputs, but do not have any mechanism for executing programs. Programmable systems are capable of executing programs. In this chapter, our focus is on nonprogrammable systems, primarily using a multiplier for illustration. Coverage of programmable systems begins in Chapter 10. The algorithmic state machine (ASM), a more user-friendly version of the state diagram for a sequential circuit, provides a representation of the behavior of the control unit, as well as the controlled register transfers. By using register transfers in the ASM, combined control unit and datapath behavior is represented.

The control units represented by ASMs in this chapter are implemented by using hardwired control. Among the specialized techniques for hardwired control design, we consider two approaches that simplify the design of larger control units when compared to the basic sequential circuit design approach in Chapter 6. Implementation of control units using microprogrammed control is touched upon very briefly.

In this chapter, the main topics are algorithmic state machines, hardwired control, and representation of algorithmic state machines using HDLs. Since the design techniques are quite general, they have an impact on most of the electronic parts in the generic computer diagram at the beginning of Chapter 1. Since the CPU and FPU in the processor chip contain significant controls for activating and sequencing register transfer operations, the processor is the major area impacted by the material in this chapter.

8-1 THE CONTROL UNIT

The binary information stored in a digital computer can be classified as either data or control information. As we saw in the previous chapter, data is manipulated in a datapath by using microoperations implemented with register transfers. These operations are implemented with adder-subtractors, shifters, registers, multiplexers, and buses. The control unit provides signals that activate the various microoperations within the datapath to perform the specified processing tasks. The control unit also determines the sequence in which the various actions are performed. Because the logic design of a digital system is often treated in two distinct parts, the register and register transfer design for datapaths was covered in Chapter 7 and the design of the control unit is covered in this chapter.

Generally, the timing of all registers in a synchronous digital system is controlled by a master clock generator. The clock pulses are applied to all flip-flops and registers in the system, including those in the control unit. To prevent clock pulses from changing the state of all registers on every clock cycle, some registers have a load control signal that enables and disables the loading of new data into the register. The binary variables that control the selection inputs of multiplexers, buses, and processing logic and the load control inputs of registers are generated by the control unit.

The control unit that generates the signals for sequencing the microoperations is a sequential circuit with states that dictate the control signals for the system. At any given time, the state of the sequential circuit activates a prescribed set of microoperations. Using status conditions and control inputs, the sequential control unit determines the next state. The digital circuit that acts as the control unit provides a sequence of signals for activating the microoperations and also determines its own next state.

Based on the overall system design, there are two distinct types of control units used in digital systems, one for a programmable system and the other for a nonprogrammable system.

In a *programmable system*, a portion of the input to the processor consists of a sequence of *instructions*. Each instruction specifies the operation that the system is to perform, which operands to use, where to place the results of the operation, and, in some cases, which instruction to execute next. For programmable systems, the instructions are usually stored in memory, either in RAM or in ROM. To execute the instructions in sequence, it is necessary to provide the memory address of the instruction to be executed. This address comes from a register called the *program counter (PC)*. As the name implies, the *PC* has logic that permits it to count. In addition, in order to change the sequence of operations using decisions based on status information from the datapath, the *PC* needs parallel load capability. So, in the case of a programmable system, the control unit contains a *PC* and associated decision logic, as well as the necessary logic to interpret the instruction. *Executing* an instruction means activating the necessary sequence of microoperations in the datapath required to perform the operation specified by the instruction.

For a *nonprogrammable system*, the control unit is not responsible for obtaining instructions from memory, nor is it responsible for sequencing the execution of

those instructions. There is no *PC* or similar register in such a system. Instead, the control unit determines the operations to be performed and the sequence of those operations, based on its inputs and the status bits from the datapath.

This chapter focuses on nonprogrammable system design. It illustrates the use of algorithmic state machines (ASMs) for control unit design plus specialized techniques for ASM implementation. Programmable systems are covered in Chapters 10 and 12.

8-2 ALGORITHMIC STATE MACHINES

A processing task can be defined by register transfer microoperations controlled by a sequencing mechanism. Such a task can be specified as a hardware algorithm that consists of a finite number of procedural steps that perform the processing task. The most challenging and creative part of digital design is the formulation of hardware algorithms that achieve the required objectives. A hardware algorithm can be used as a basis for both the datapath and the control unit of a system.

A flowchart is a convenient way to specify a sequence of procedural steps and decision paths for an algorithm. A flowchart for a hardware algorithm must have special characteristics that tie it closely to the hardware implementation of the algorithm. As a consequence, we use a special flowchart called an *algorithmic state machine* (ASM) chart to define digital hardware algorithms. A *state machine* is just another term for a sequential circuit.

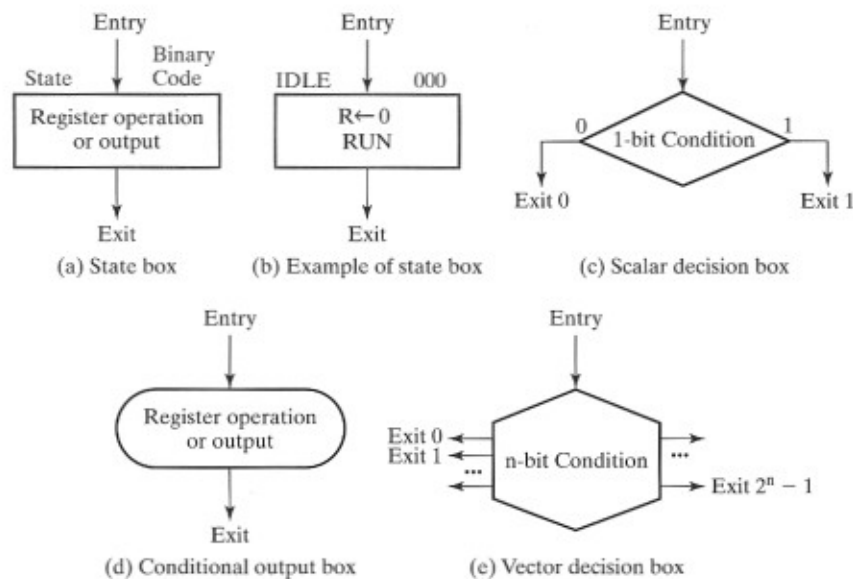
The ASM chart resembles a conventional flowchart, but is interpreted somewhat differently. A conventional flowchart describes procedural steps and decision paths without any concern for their relationship to time. By contrast, the ASM chart provides not only a sequence of events, but is distinguished by the fact that it describes the timing relationship between the states of the control unit and the datapath actions that occur in the states in response to clock pulses,

The ASM Chart

The ASM chart contains three basic elements: the state box, the scalar decision box, and the conditional output box, as illustrated in Figure 8-1. For convenience, a fourth element, the vector decision box, has been added. This additional component simplifies representation of multiway decisions and establishes a correspondence between HDL representations and ASM charts.

A state in the control sequence is indicated by a state box, as shown in Figure 8-1(a). The *state box* is a rectangle containing register transfer operations or output signals that are activated while the control unit is in the state. Implicitly, activation of an output signal means assigning a value of 1 to the signal. The symbolic name for the state is placed at the upper left corner of the box, and the binary code for the state, if assigned, is placed at the upper right corner of the box.

Figure 8-1(b) shows a specific example of a state box. The state has the symbolic name *IDLE*, and the binary code assigned to it is 000. Inside the box is the register transfer $R \leftarrow 0$ and the output *RUN*. The register transfer indicates that the



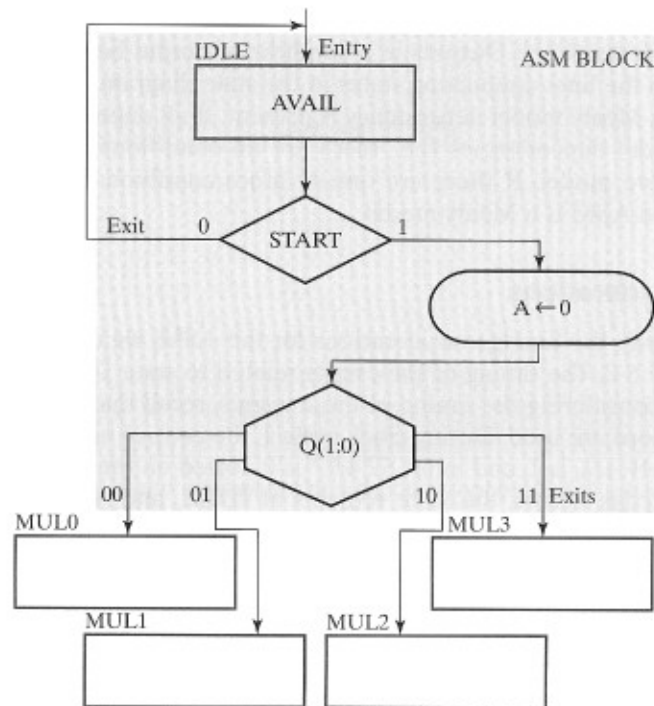
□ **FIGURE 8-1**
ASM Chart Elements

register R is to be reset to 0 on any clock pulse that occurs while the control is in state *IDLE*. *RUN* indicates that the output signal *RUN* is to be 1 during the time that the control is in state *IDLE*. *RUN* is 1 for any state box in which it appears and is 0 for any state box in which it does not appear.

The *scalar decision box* describes the effect of an input on the control. It is a diamond-shaped box with two exit paths, as shown in Figure 8-1(c). The input condition is a single binary input variable or a single Boolean expression dependent only upon inputs. One exit path is taken if the input condition is true (1), and the other is taken if the input condition is false (0).

The third element, the *conditional output box*, is unique to the ASM chart. The oval shape of the box is shown in Figure 8-1(d). The rounded corners differentiate it from the state box. The entry path to a conditional output box from a state box must pass through one or more decision boxes. If the conditions specified on the path through the decision boxes leading from the state box to a conditional output box are satisfied, the register transfers or outputs listed inside the conditional output box are activated.

The *vector decision box* shown in Figure 8-1(e) describes the effect of a vector function of inputs on the control. It is a hexagon-shaped box with up to 2^n exit paths for an n -element binary vector. The input condition is a vector of $n > 1$ binary input variables or Boolean expressions dependent upon only the inputs. An exit path is taken if the vector value matches the label corresponding to the exit path.



□ **FIGURE 8-2**
ASM Block

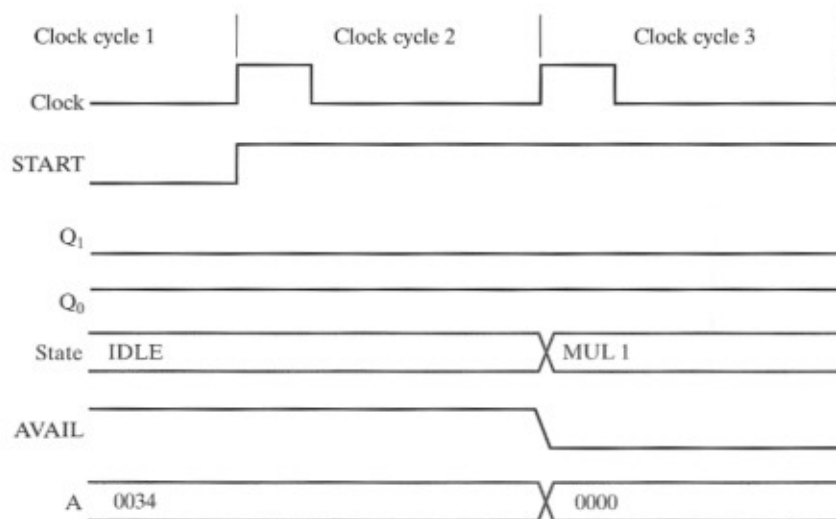
An *ASM block* consists of one state box and all of the decision and conditional output boxes connected between the state box exit and entry paths to the same or other state boxes. An example of an ASM block is shown in Figure 8-2. The block represents decisions and output actions that can take place in the state. Any outputs for which conditions are satisfied within the ASM block are activated in the block. Any register transfers for which conditions are satisfied within the ASM block will be executed when the clock event occurs. This same clock event will also transfer control to the next state as specified by decisions within the ASM block. For the block in Figure 8-2, the state is *IDLE*. While in the state *IDLE*, the output *AVAIL* is equal to 1. If *START* is 0, then the next state is *IDLE*. If *START* is 1, then at the clock event, *A* is cleared to all 0's, and, depending on the value of the vector *Q(1:0)*, the next state is *MUL0*, *MUL1*, *MUL2*, or *MUL3*. In the figure, the entry path and the five exit paths for the ASM block are labeled at the boundaries of the ASM block.

The ASM chart is really a form of state diagram for the sequential circuit part of the control unit. Each state box is equivalent to a node in the state diagram. The decision boxes are equivalent to input values on the lines that connect nodes in the diagram. The register transfers and outputs in the state boxes and the conditional output boxes correspond to the outputs of the sequential circuit. Outputs in a state

box are those that would be specified on a state node in the state diagram with a Moore model dependency. Outputs in a conditional output box correspond to the input values on the lines connecting states in the state diagram. Since these depend on the inputs, a Mealy model dependency is present. If all dependencies in an ASM are Moore model dependencies (i.e., there are no conditional output boxes), the ASM is a Moore model. If there are one or more conditional boxes with Mealy dependency, the ASM is a Mealy model.

Timing Considerations

In order to clarify the timing considerations for the ASM, we use the sample ASM block in Figure 8-2. The timing of the events related to state IDLE is illustrated in Figure 8-3. In considering the timing of these events, recall that only positive-edge-triggered flip-flops are used. During clock cycle 1, the control unit is in present state IDLE, output *AVAIL* is 1, and input *START* is 0. Based on the ASM block, when a positive clock edge occurs, the state remains at IDLE, and *AVAIL* remains at 1. Also, the contents of register *A* remain unchanged. In clock cycle 2, *START* becomes 1. So when the next positive clock edge occurs, register *A* is cleared to 0. With *START* at 1, *Q*(1:0) is examined and found to be 01. For this value, when the clock edge occurs, the next state becomes MUL1. The new state MUL1 and the new value of *A* both appear at the beginning of clock cycle 3. The value of *AVAIL* becomes 0, since *AVAIL* does not appear in the state box for state MUL1. Note that the output *AVAIL* = 1 appears concurrently with the present state IDLE, but the result of the register transfer for *A* appears concurrently with the next state MUL1. This is because outputs occur asynchronously in response to state and input values, but register transfers and state changes both wait until the next positive clock edge.



□ **FIGURE 8-3**
ASM Timing Behavior

23	10111	Multiplicand
<u>19</u>	<u>10011</u>	Multiplier
	10111	
	10111	
	00000	
	00000	
	<u>10111</u>	
437	110110101	Product

□ **FIGURE 8-4**
Hand Multiplication Example

8-3 ASM CHART EXAMPLES

A binary multiplier is used to illustrate ASM chart formulation. The multiplier multiplies two n -bit unsigned integers to produce a $2n$ -bit integer result.

Binary Multiplier

In this example, we introduce a hardware algorithm for binary multiplication, propose a simple datapath and control unit for its implementation, and then describe its register transfers and control by use of an ASM. The system used for illustration multiplies two unsigned binary numbers. In Section 5-5, a hardware algorithm to execute this multiplication without using storage elements resulted in a combinational multiplier with many adders and AND gates. In contrast, the hardware algorithm developed here results in a sequential multiplier that uses only one adder and a long shift register. The algorithm is illustrated, the register transfer structure proposed, and the ASM chart formulated.

MULTIPLICATION ALGORITHM The multiplication of two unsigned binary numbers is done with paper and pencil by successive shifts of copies of the multiplicand to the left and an addition. The process is best illustrated with an actual example. Let us multiply the two binary numbers 10111 and 10011, as shown in Figure 8-4. To carry out the multiplication, we look at successive bits of the multiplier, least significant bit first. If the multiplier bit is 1, the multiplicand is copied down for use in the addition to follow. Otherwise 0's are copied down. The numbers copied in successive lines are shifted one position to the left from the previous number copied, to align them with the respective multiplier bit being processed. Finally, the numbers are added and their sum forms the product. Note that the product obtained by multiplying two n -bit binary numbers can have up to $2n$ bits for $n \geq 2$.

When the multiplication procedure is implemented with digital hardware, it is useful to change the process slightly. First, instead of having a digital circuit that adds n binary numbers simultaneously, it is less expensive to provide a circuit that adds just two numbers. Each time the multiplicand or 0's are copied, they are immediately added to a *partial product*. The partial product is stored in a register in

preparation for the shift action to follow. Second, instead of shifting the copies of the multiplicand to the left, the partial product is shifted to the right. This leaves the partial product and the copy of the multiplicand in the same relative position as the left shift of the multiplicand did. But, more important, instead of a $2n$ -bit adder, only an n -bit adder is needed. The addition always takes place in the same n positions, instead of moving to the left one bit position each time. Third, when the corresponding bit in the multiplier is 0, there is no need to add all 0's to the partial product, since this does not alter its resulting value.

The multiplication example is repeated in Figure 8-5 with these changes. Note that the initial partial product is 0. Each time the multiplier bit being processed is 1, an addition of the multiplicand, followed by a right shift, is performed. Each time the multiplier bit is a 0, only a right shift is performed. One of these two actions is performed for each bit of the multiplier, so in this case, five such actions occur. An unsigned overflow occurring during an addition is indicated in blue. This overflow is not a problem, however, since the right shift that immediately follows brings the extra partial product bit into the regular most significant bit position.

MULTIPLIER BLOCK DIAGRAM The block diagram for the binary multiplier is shown in Figure 8-6. The multiplier datapath is first constructed from components covered in previous chapters. All but counter P are expanded to n bits; counter P requires $\lceil \log_2 n \rceil$ bits for counting the processing of the n bits of the multiplier. ($\lceil x \rceil$ denotes the smallest integer greater than or equal to x .) We use the parallel adder from Figure 5-5, a parallel-load register B similar to Figure 7-2, and parallel-load shift registers A and Q similar to Figure 7-10. Counter P is a version of the parallel-load

23	10111	Multiplicand
<u>19</u>	<u>10011</u>	Multiplier
	00000	Initial partial product
	<u>10111</u>	Add multiplicand, since multiplier bit is 1
	10111	Partial product after add and before shift
	010111	Partial product after shift
	<u>10111</u>	Add multiplicand, since multiplier bit is 1
	1000101	Partial product after add and before shift ^a
	1000101	Partial product after shift
	01000101	Partial product after shift
	001000101	Partial product after shift
	<u>10111</u>	Add multiplicand, since multiplier bit is 1
	110110101	Partial product after add and before shift
437	0110110101	Product after final shift

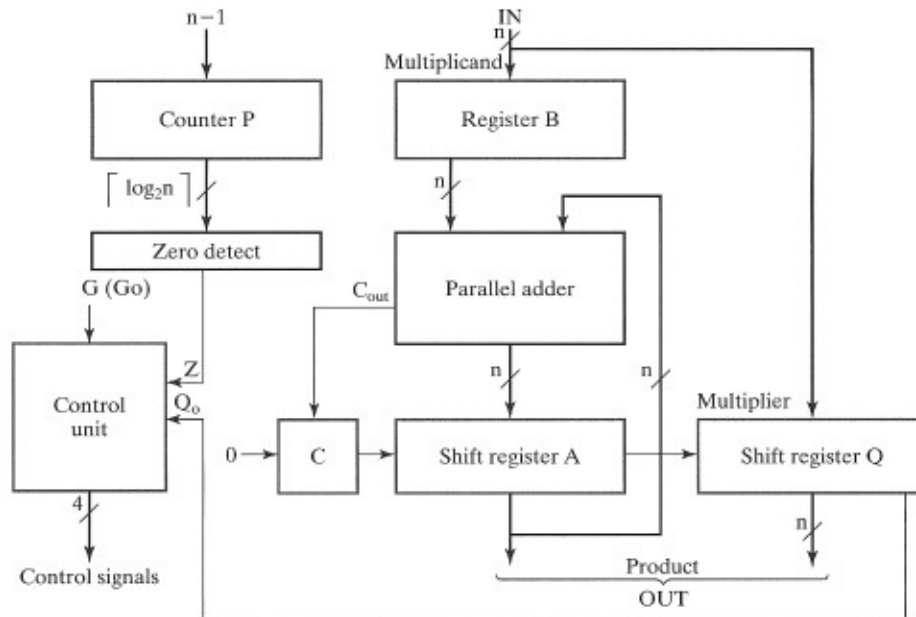
a. Note that overflow temporarily occurred.

□ **FIGURE 8-5**
Hardware Multiplication Example

counter in Figure 7-14 that counts down instead of up, and C is a flip-flop that can be either synchronously cleared to 0 or loaded from C_{out} . These datapath components are connected as shown in Figure 8-6.

The multiplicand is loaded into register B from IN , the multiplier is loaded into register Q from IN , and the partial product is formed in register A and stored in registers A and Q . This dual use of Q is possible because we use a right shift of the multiplier in Q to examine each successive multiplier bit that appears in Q_0 . The right shift vacates the most significant bit in register Q . This space accepts the least significant bit of the partial product from A as it is shifted. The n -bit binary adder is used for adding B to A . The C flip-flop stores the carry C_{out} , whether 0 or 1, from the addition and is reset to 0 during the right shift. In order to count the number of add-shift or shift actions that are to occur, counter P is provided. It is initially set to $n - 1$ and counted down after the formation of each partial product. The value in P is checked just before it is decremented. So n operations occur, one operation for each value in P , $n - 1$ down through 0. Each operation is either an add and shift or just a shift. When P contains 0, the final product is in the double register A and Q , and processing stops.

The control unit stays in an initial state until the Go signal G becomes 1. Then the system starts the multiplication. The sum of A and B forms the n most significant bits of the partial product, which is transferred back to A . C_{out} from the addition is transferred to C . Both the partial product and the multiplier in A and Q are shifted to the right. The carry from C shifts into the most significant bit of A ,



□ **FIGURE 8-6**
Block Diagram for Binary Multiplier

the least significant bit of A shifts into the most significant bit of Q , and the least significant bit of Q is discarded. After this right-shift operation, one (additional) bit of the partial product has transferred into Q , and the multiplier bits have shifted one position to the right. In this manner, the least significant bit of Q , Q_0 , always holds the bit of the multiplier that the control unit examines next. The control unit “decides” whether to add, based on the value of this bit. It also checks signal Z , which is 1 for P equal to zero and 0 for P nonzero, to determine whether the multiplication is finished. Q_0 and Z are the status inputs for the control unit, with input G as the only external control input. The control signals from the control unit to the datapath activate the required microoperations.

MULTIPLIER ASM CHART An ASM chart giving the sequence of operations in the binary multiplier is shown in Figure 8-7. Initially, the multiplicand is in B and the multiplier in Q . The loading of these two registers is not handled explicitly by the multiplier control unit. As long as the ASM is in state IDLE, and G is 0, no actions occur, and the ASM remains in IDLE. The multiplication process starts when G becomes 1. As the ASM moves from state IDLE to state MUL0, registers C and A are cleared to 0, and the counter P is loaded with the constant $n - 1$. In state MUL0, a decision is made based upon Q_0 , the least significant bit of Q . If Q_0 is 1, the contents of B are added to those of A , with the result transferred to A and the carry transferred to C . If Q_0 is 0, register A and bit C are left unchanged. In both cases, the next state is MUL1.

In state MUL1, a right shift is performed on the combined contents of C , A , and Q . This shift can be expressed by the somewhat messy list of five simultaneous register transfers:

$$C \leftarrow 0, A(n-1) \leftarrow C, A \leftarrow \text{sr } A, Q(n-1) \leftarrow A(0), Q \leftarrow \text{sr } Q$$

To simplify representation of this operation, we add a bit of notation, using \parallel to define a *composite register* made up of other registers or pieces of other registers. This operation, \parallel , is called *concatenation*. For example,

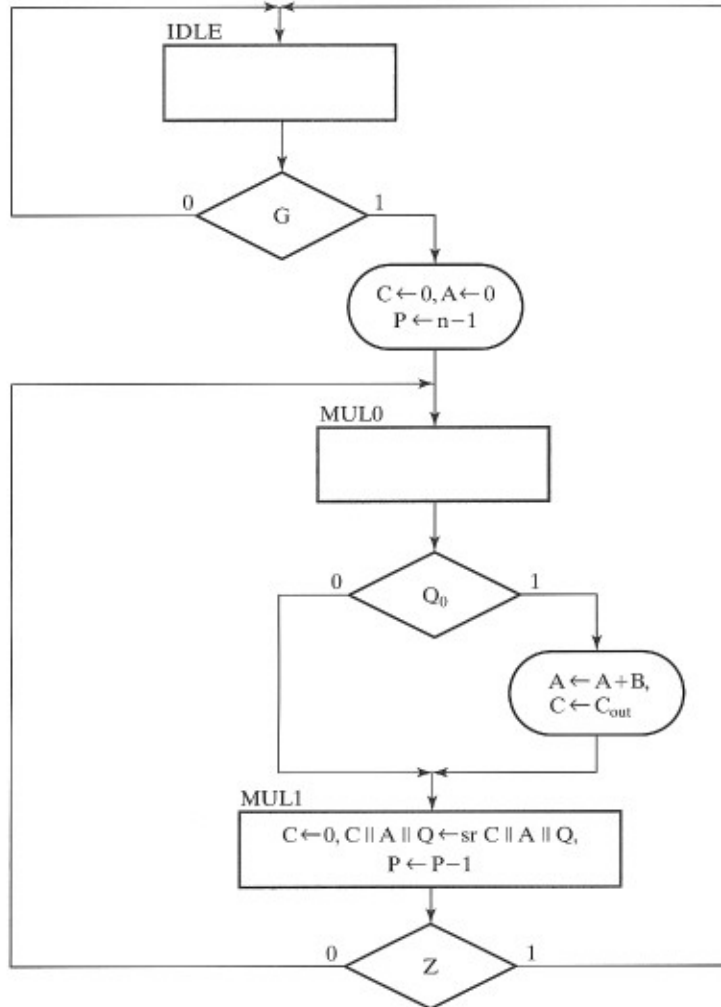
$$C \parallel A \parallel Q$$

represents a single register obtained by combining registers C , A , and Q from the most significant end to the least significant end. We can use this composite register to represent the right shift

$$C \parallel A \parallel Q \leftarrow \text{sr } C \parallel A \parallel Q$$

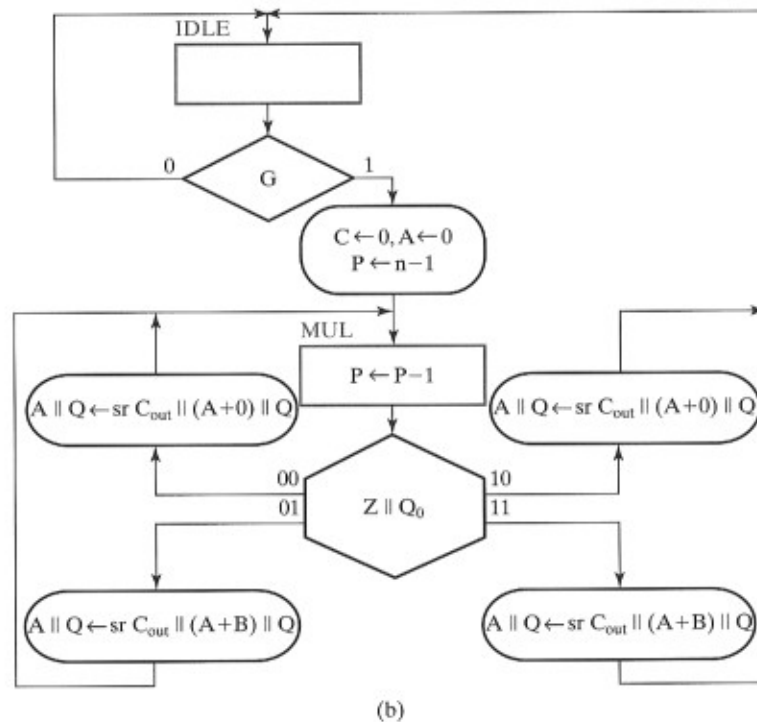
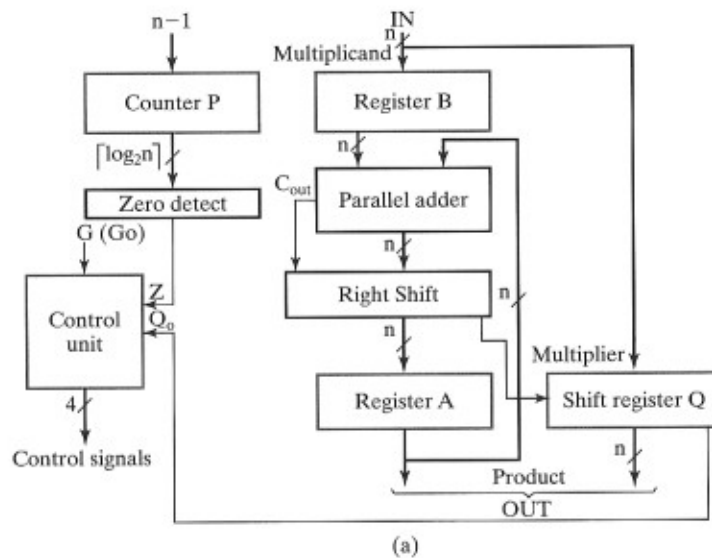
as shown in Figure 8-7. Recall that we are assuming that the leftmost bit of the result for a right shift takes on the value 0 unless otherwise specified, so C becomes 0. This is represented explicitly, however, in the ASM chart, since C is set to 0 in another state as well. The explicit listing allows $C \leftarrow 0$ to be performed by using a single control signal for both states.

Counter P is decremented in MUL1. The value in P is checked in state MUL1 before P is decremented. This illustrates a very important timing difference between a standard flowchart and an ASM chart. The decision on Z , which represents $P = 0$, follows the register transfer statement that updates P in the ASM



□ **FIGURE 8-7**
ASM Chart for Binary Multiplier

chart. Since the decision on P is performed asynchronously and the register transfer statement is synchronous with the next positive clock edge, the decision on P precedes the update of P . At the next clock edge, when P is updated, the result of this decision is available to determine the next state. The first $n - 1$ times that P is checked, its content is nonzero, so status bit Z remains 0, and the loop, consisting of states MUL0 and MUL1, is executed again. The n th time P is checked, the content of P is zero, so status bit Z is 1. This indicates that the multiplication is complete, causing the ASM to return to state IDLE. The final product is available in $A \parallel Q$, with A holding the n most significant bits and Q the n least significant bits of



□ **FIGURE 8-8**
Alternative Binary Multiplier

the product. It is worthwhile to reexamine the hardware multiplication example for $n = 5$ in Figure 8-5, this time considering the relationship to the datapath and the flow of the ASM chart.

The type of registers selected for the datapath correspond to the microoperations listed in the ASM chart. Register A is a shift register with parallel load that accepts the sum from the adder. It also needs a synchronous clear to reset the register to 0. Register Q is a shift register. The C flip-flop needs to accept the input carry and also needs a synchronous clear. Registers B and Q also need parallel load in order to load the multiplicand and the multiplier prior to the start of the multiplication process.

Figure 8-8 shows an alternative multiplier design that uses the vector decision box in its ASM chart. In part (a) of the figure, Shift Register A has been replaced by a combinational right shifter that shifts 1 bit to the right, similar to Figure 5-13(c), and a register with load enable as shown in Figure 7-2. This combines the adder with a combinational right shift. This permits the number of states used in the ASM chart in Figure 8-8 to be reduced to just one. In order to represent the change in the multiplier datapath, it is necessary to write a register transfer statement that combines the addition with the shift. Also, the datapath change permits the flip-flop C to be deleted. Assuming that the combined delay of the adder and right shift (which consists only of wires) is no more than the adder, the reduction in states in the multiply loop substantially speeds up the multiplication operation.

To illustrate the vector decision box, we have used concatenation to combine Z and Q_0 into the vector (Z, Q_0) which is denoted as $Z \parallel Q_0$. The decision based on this vector is shown in the center of the ASM chart in Figure 8-8(b). There are four output combinations. For the combinations in which Z is 1, the next state is IDLE. For the combinations in which $Z = 0$, the next state is MUL. For the combinations in which $Q_0 = 1$, the output is an add-right shift with input operands A and B , and for the combinations in which $Q_0 = 0$, the output is an add-right shift with input operands A and 0. These are represented by the combined add and shift transfers in conditional output boxes for the four output combinations for $Z \parallel Q_0$.

8-4 HARDWIRED CONTROL

In implementing a control unit, two distinct aspects must be considered: the control of the microoperations and the sequencing of the control unit and microoperations. Very simply put, the first has to do with the part of the control that generates the control signals, and the second has to do with the part of the control that determines what happens next. Here, we separate these two aspects by dividing the original ASM specification into two parts: a table that defines the control signals in terms of states and inputs, and a simplified ASM chart that represents only transitions from state to state. Although we are separating these two aspects for design purposes, they can share logic.

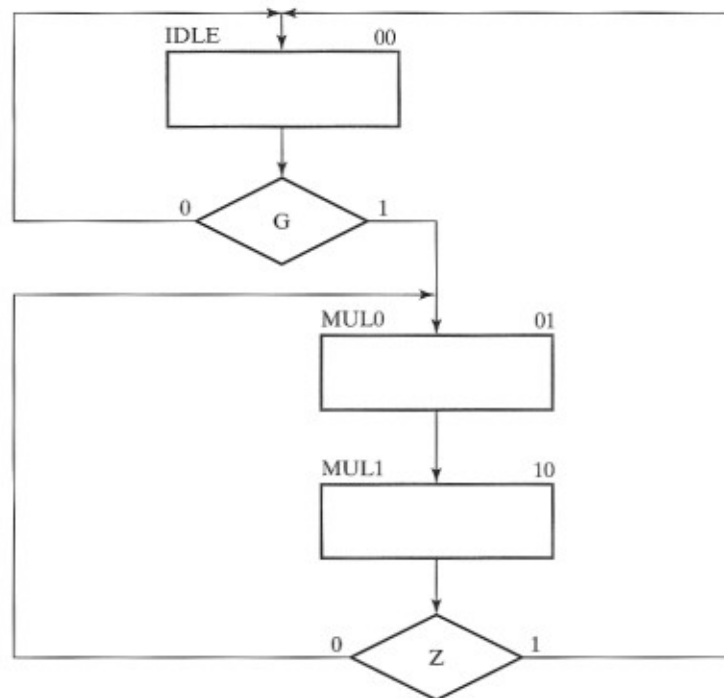
The control signals are based on the ASM chart. The control signals needed for the multiplier datapath are listed in Table 8-1, where we have chosen to examine the datapath registers and tabulate the microoperations for each register. Based on the tabulated microoperations, the control signals are defined. A control

□ TABLE 8-1
Control Signals for Binary Multiplier

Block Diagram Module	Microoperation	Control Signal Name	Control Expression
Register <i>A</i> :	$A \leftarrow 0$	Initialize	$IDLE \cdot G$
	$A \leftarrow A + B$	Load	$MUL0 \cdot Q$
	$C \parallel A \parallel Q \leftarrow sr C \parallel A \parallel Q$	Shift_dec	MUL1
Register <i>B</i> :	$B \leftarrow IN$	Load_B	LOADB
Flip-Flop <i>C</i> :	$C \leftarrow 0$	Clear_C	$IDLE \cdot G + MUL1$
	$C \leftarrow C_{out}$	Load	—
Register <i>Q</i> :	$Q \leftarrow IN$	Load_Q	LOADQ
	$C \parallel A \parallel Q \leftarrow sr C \parallel A \parallel Q$	Shift_dec	—
Counter <i>P</i> :	$P \leftarrow n - 1$	Initialize	—
	$P \leftarrow P - 1$	Shift_dec	—

signal can be used for activating microoperations in more than one register. This is reasonable, in this case, since the datapath is dedicated to only one operation, multiplication. Thus, the control signals do not need to be separated to provide the generality required for implementing additional, potentially unknown operations. Finally, the Boolean expression for each control signal is derived from the location or locations of the microoperation in the ASM chart. For example, for register *A*, there are three microoperations shown in Table 8-1: clear, add and load, and right shift. Since the clear operation always occurs at the same time as the clear for flip-flop *C* and the loading of counter *P*, all of these microoperations can be activated by the same control signal, named Initialize. Because *C* is cleared in state MUL1 as well, however, we choose to separate its control signal. So Initialize is used for clearing *A* and loading *P*. In the last column for Initialize, the Boolean expression for which Initialize is to be active, as determined from the ASM chart, is given in terms of the state IDLE and input *G*. Since Initialize is to be 1 when *G* is 1 in state IDLE, IDLE and *G* are ANDed. At this point, the name for the state is treated as a Boolean variable. Depending on the implementation, there may be such a signal representing the state, or the state may need to be expressed as a function of the state variables. The signal for clearing *C*, Clear_C, is to be active in state IDLE for *G* equal to 1, as well as in state MUL1. Thus, *G* is ANDed with IDLE, and the result is ORed with MUL1. The other two internal multiplier control signals, Load and Shift_Dec, are defined in a similar manner. The final two signals, Load_B and Load_Q, load the multiplicand and multiplier from outside the multiplier system. These signals will not be considered explicitly in the remainder of the design.

With the information on microoperations removed, we can redraw the ASM chart so that only the information on sequencing is represented. This modified ASM chart for the binary multiplier appears in Figure 8-9. Note that all of the conditional output boxes have been removed. In addition, any decision box not affecting the next state is removed. In particular, in Figure 8-7, the decision box Q_0



□ **FIGURE 8-9**
Sequencing Part of ASM Chart for the Binary Multiplier

affected only a conditional output box. Once that conditional output box is removed, the two exit paths from decision box Q_0 clearly go to the same state. So this decision box has no effect on the next state and is removed.

From this modified ASM chart, we can design the sequencing part of the control unit (i.e., the part that represents the next-state behavior). The division of control into next-state behavior in the form of the modified ASM chart and output behavior in the form of the control signal table shows how the ASM corresponds to the next-state and output parts of a sequential circuit. Figure 8-9 corresponds to the state diagram of a sequential circuit without the outputs specified, except that the representations used in the diagram for states and transitions are different. Because of this correspondence, we can treat the ASM chart as a state diagram and form a state table for the sequencing part of the control unit. Then the control unit can be designed by the sequential logic design procedure, as outlined in Chapter 4. However, in many cases, this method is difficult to carry out because of the large number of states for a typical control unit. As a consequence, we use specialized methods for control unit design that are variations of the classical sequential logic methods. We next present and illustrate two such design methods.

Sequence Register and Decoder

The sequence register and decoder method, as the name implies, uses a sequence register for the control states and a decoder to provide an output signal corresponding to each of the states. A register with n flip-flops can have up to 2^n states and an n -to- 2^n decoder has up to 2^n outputs, one for each of the states. An n -bit sequence register is essentially n flip-flops, together with the associated gates that effect their state transitions. Additional logic may be required to produce the necessary control signal outputs.

The sequencing part of the ASM chart for the binary multiplier has three states and two inputs. To implement the ASM chart with a sequence register and decoder, we need two flip-flops for the register and a 2-to-4-line decoder. Since there are three states, only three of the four decoder outputs are used. Although this is a simple example, the procedure to be outlined applies to more complex situations as well.

The state table for the sequencing part of the control unit is shown in Table 8-2; it is derived directly from the ASM chart in Figure 8-9. We designate the two flip-flops as M_1 and M_0 and assign the binary states 00, 01, and 10 to IDLE, MUL0, and MUL1, respectively. Note that the input columns have unspecified entries (\times) whenever an input variable is not used to determine the next state. The outputs of the sequencing part of the control are designated by the state names. The binary code for the present state determines the particular output variable that is equal to 1 at any given time. Thus, when the present state is $M_1M_0 = 00$, output IDLE equals 1, while the other outputs equal 0. Since these outputs depend on the present state only, they can be generated with the 2-to-4-line decoder having inputs M_1 and M_0 and outputs IDLE, MUL0, and MUL1.

As mentioned earlier, the sequential circuit can be designed from the state table by means of the sequential logic design procedure presented in Chapter 4. This example has a small number of states and inputs, so we could use maps to simplify the Boolean functions. In most control logic applications, however, the number of states is much larger. The application of the conventional method

□ **TABLE 8-2**
State Table for Sequence Register and Decoder Part
of Multiplier Control Unit